EFFICIENT PARALLEL DIGITAL SIGNAL PROCESSING ALGORITHMS FOR HYPERCUBE CONNECTED MULTICOMPUTERS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING AND INFORMATION SCIENCE AND THE INSTITUTE OF ENGINEERING AND SCIENCE OF BILKENT UNIVERSITY IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

By

Argun DERVİŞ

April 1992

EFFICIENT PARALLEL DIGITAL SIGNAL PROCESSING ALGORITHMS FOR HYPERCUBE-CONNECTED MULTICOMPUTERS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER

ENGINEERING AND INFORMATION SCIENCE

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Argun Derviş

April 1992

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Cevdet Aykanat(Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Kemal Oflazer

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Enis Çetin

Approved by the Institute of Engineering and Science:

Prof. Mehmet Baray, Director of the Institute of Engineering and Science

ABSTRACT

EFFICIENT PARALLEL DIGITAL SIGNAL PROCESSING ALGORITHMS FOR HYPERCUBE-CONNECTED MULTICOMPUTERS

Argun Derviş
M. S. in Computer Engineering and Information Science
Supervisor: Assoc. Prof. Cevdet Aykanat
April 1992

In this thesis, efficient parallelization of Digital Signal Processing (DSP) algorithms, (FFT, FHT and FCT), on multicomputers implementing the hypercube interconnection topology are investigated. The proposed algorithms, maintain perfect load-balance, minimize communication overhead, can overlap communications with computations and achieve regular computational patterns. The proposed parallel algorithms are implemented on Intel's iPSC/2[1] hypercube multicomputer with 32 processors. High efficiency and almost linear speedup values are obtained for even small size problems.

Keywords: Digital Signal Processing, Hypercube, Parallel Processing, FFT, FHT, FCT.

[1]iPSC/2 is a registered trademark of Intel Corporation

ÖZET

HİPERKÜP ÇOK İŞLEMCİLİ BİLGİSAYARLARINDA VERİMLİ PARALEL SAYISAL İŞARET İŞLEME ALGORİTMALARI

Argun Derviş
Bilgisayar Mühendisliği ve Enformatik Bilimleri Bölümü
Yüksek Lisans
Tez Yöneticisi: Assoc. Prof. Cevdet Aykanat
Nisan 1992

Bu tez kapsaminda, hiperküp bağlanti yapisini içeren çok işlemcili bilgisayarlarda, tek boyutlu Sayisal İşaret İşleme algoritmalari, FFT, FHT ve FCT araştirilmiştir. Önerilen algoritmalar, eşit yük dağilimi, minimum haberleşme, haberleşmeleri sayisal işlemlerle birleştirebilme ve düzgün algoritmik yapilar içermektedir. Önerilen paralel algoritmalar, Intel'in hiperküp bilgisayarinda, 32 işlemcisiyle denenmiştir. Küçük boyuttaki problemler için bile yüksek kazanç ve hizlar elde edilmiştir.

Anahtar kelimeler : Sayisal İşaret İşleme, Hiperküp, Paralel İşleme, FFT, FHT, FCT.

ACKNOWLEDGEMENT

I would like to express my deepest gratitude to Assoc. Prof. Dr. Cevdet Aykanat for his guidance, suggestions, advice, encouragement and his friendly supervision. I shall always be indebted to ASELSAN Military Electronics Inc. and the manager of Embedded Systems Division İsmet Atalar for the considerable support that, they have provided to me during my study.

I am also gratefull, to Assoc. Prof. Dr. Kemal Oflazer and Assoc. Prof. Dr. Enis Çetin for their valuable suggestions. I would also like to thank to my friends especially to Taçli Yazicioğlu and Ayşegül Göçmen for their endless support and morale.

I finally would like to thank to my family for their endless support and kindness that they have provided to me.

Contents

List of Figures

List of Tables

1. Introduction

In the 1960s, a group of engineers and mathematicians working at AT&T Bell Labs invented a set of techniques known as *Digital Signal Processing* (DSP). The algorithms they invented are used today in many fields, including radar, sonar, seismic processing, image enhancement, communications, radio astronomy, medical imaging and etc.

Digital Signal Processing of real-time signals has gained importance with recent advances in digital computer technology. Digital signal processors - digital computers specializing in signal processing - are in development and available on the market. All of this growth is for massive amounts of computations in various DSP applications.

One way to satisfy the performance requirement of DSP applications is to choose clever algorithms or expand the processor performance or both of them. DSP applications are characterized by computations that are massive but fairly straightforward and simple. Furthermore, these computations exhibit orderly structures. Besides, DSP algorithms are very efficient. These algorithms are optimized and improved several times until now. However, it is still not enough for most of the DSP applications. Performance of conventional computers are still very limited in cases where extensive number crunching computations are required. *Discrete Fourier Transform* (DFT), *Discrete Hartley Transform* (DHT) and *Discrete Cosine Transform* (DCT) are such examples.

Discrete Fourier Transform provides an efficient method for spectral analysis of discrete signals. Thus, Cooley and Tukey providing a more efficient algorithm [1], named as *Fast Fourier Transform* (FFT), made possible many applications concerning the computation of DFT to be realizable because of performance problems. FFT algorithm has a very wide application domain and known to be the most widely used digital signal processing transformation.

Beyond the highly accepted usage of FFT, it is a complex transformation. If the signals in the time domain are real, then the FFT contains redundancy. *Discrete Hartley Transform* (DHT) exists for this reason [11] and can be called as a real version of DFT. *Discrete Hartley Transform* is applied to signals, where the domain is formed of only real inputs. As well as FFT, *Discrete Hartley Transform* has also a fast formulation called *Fast Hartley Transform* FHT [13]. FHT provides efficient spectral analysis of real discrete signals and it is faster than FFT.

Dicrete Cosine Transform (DCT) [20] is another transformation for DSP. DCT is used mainly for speech and image coding applications. DCT is also a real transformation like DHT and has also a fast computation formula referred as *Fast Cosine Transform* [25] or shortly FCT in this work. Importance of FCT comes from the fact that, it has the best performance among the other known fast computable transforms. With this feature FCT is a promising alternative to FFT and FHT.

As mentioned earlier, although fast algorithms exist for DSP transformations, conventional computers are still not fast enough to compute these algorithms (FFT, FHT, FCT) in real-time. Extensive research has been made in this area, in order to overcome the difficulties faced during real-time computation of DSP algorithms. This led people to design special purpose hardwares, DSP microprocessors, VLSI circuits and use general high-performance computers to solve these problems in real-time. Distributed memory, message-passing parallel computers, which are usually named as multicomputers, are

the most promising general purpose high performance computers. These architectures have the nice scalability feature due to the lack of shared resources and increasing communication bandwidth with the increasing number of processors. In multicomputers, processors have neither shared memory nor shared address space. Each processor can only access its local memory. Processors of a multicomputer are usually connected by utilizing one of the well-known direct interconnection network topologies such as ring, mesh, hypercube and etc. Hypercube topology is very well suited for many of the DSP applications including FFT, FHT and FCT.

In this thesis, efficient parallelization of several one-dimensional *FFT, FHT* and *FCT* algorithms on multicomputers implementing the hypercube interconnection topology are investigated. In order to achieve speedup on such architectures, the algorithm must be designed so that both computations and data can be distributed to the processors with local memories in such a way that computational tasks can be run in parallel, balancing the computational loads of the processors. Communication between processors to exchange data must also be considered as part of the algorithm. One important factor in designing parallel algorithms is *granularity. Granularity* depends on both the application and the parallel machine. In a parallel machine with high communication latency, the algorithm should be structured so that large amounts of computation are done between successive communication steps. Another important factor is the ability of the parallel system to *overlap* communication and computation. In order to exploit this property of the parallel system, the algorithm must be structured so that the communication can be overlapped with computation. The algorithms presented in the following chapters, achieve efficient parallelization by considering all these points in designing efficient parallel one dimensional *FFT, FHT* and *FCT* algorithms for hypercube multicomputers. These parallel algorithms eliminate fragmentary message passing for efficient parallelization on medium-to-coarse grain, MIMD type hypercube multicomputers. Proposed parallel algorithms do not disturb the inherent regularity of FFT, FHT and FCT so that they can be implemented on SIMD type hypercube multicomputers. These algorithms are structured such that overlapping communications with computations is possible. Proposed algorithms

are written and compiled in ANSI C language on a 32-node iPSC/2 hypercube multicomputer.

In chapter 2.3.1, perfect-load balance FFT algorithm is discussed. In chapter 2.3.2, overlapping FFT algorithms are presented. In chapter 2.4, experimental results are given for the presented algorithms.

In chapter 3.3.1, restructuring of FHT along with its perfect-load balance algorithm is presented. In chapter 3.4, experimental results are given for the presented algorithms.

In chapter 4.2.2, parallelization of FCT along with its relation to FHT is presented. In chapter 4.3, perfect-load balance FCT algorithm is given. In chapter 4.4, experimental results are given for the presented algorithms.

2. The Fast Fourier Transform

2.1 Introduction

After its' foundation in the 19^{th} Century by Jean Baptiste Joseph Fourier, Discrete Fourier Transform (DFT) has been the most widely used signal processing transform all over the world. It simply transforms signals from their time domain to their frequency domain; in other words, obtains the frequency components of a signal.

Discrete Fourier Transform is the heart of digital signal processing applications and used in most of the digital signal processing applications. Some include cellular-phones, radar, sonar, satellite communications, VCRs, hi-fi equipment, camcorders etc., Also other DSP transforms can also be computed with the help of DFT.

The Discrete Fourier Transform of a signal $f(i)$ $(i = 0, 1, \ldots, N-1)$ is,

$$\begin{aligned} F(k) &= \sum_{i=0}^{N-1} f(i) W_N^{ki} \\ &= \sum_{i=0}^{N-1} (f(i)[Cos(2\pi ki/N) - jSin(2\pi ki/N)]) \end{aligned} \tag{2.1}$$

where (k=0,1,..., N-1) and $W_N^{ki} = e^{-j2\pi ki/N}$.

As is written, DFT requires on the order of N^2 multiplications and N^2 additions. If N is a power of 2, (eg., $N = 2^n$) then there are several ways to change

this Fourier Transform into something different called Fast Fourier Transform (FFT) which can be calculated in less time [1]. The difference between DFT and FFT is the computation methods.

Different strategies exist for the computation of FFT and some include decimation in-time, decimation in-frequency, radix-2, radix-4, radix-8, mixed-radix etc. Some algorithms are named after their founders such as Cooley-Tukey FFT [1], Good-Thomas FFT [2], Goertzel FFT [4], Winograd FFT [3] etc.

The *FFT* scheme choosen for parallelization is Radix-2 Cooley-Tukey scheme and the sequential algorithm is presented and discussed in Section 2.2. Parallelization of the chosen *FFT* scheme is discussed in Section 2.3. Parallel *FFT* algorithms presented in Section 2.3 are implemented on an 32-node iPSC/2 hypercube multicomputer. Section 2.4 presents the implementation results and the relative experimental performance evaluation of the designed algorithms.

2.2 The Sequential FFT Algorithm

There are many variants of the *FFT* algorithm in the literature [5]. The *decimation-in-time* decomposition scheme, with input in *bit-reversed* order, is investigated in this chapter. The computational flow graph of this scheme is given in Figure 2.3 for a (N=16)-point FFT computation. The input in this scheme is N-complex numbers in *bit-reversed* order. The output is N-complex numbers in *normal order*. As is seen in Figure 2.3, each stage of the computation takes a set of N complex numbers and transforms them into another N complex numbers. This process is repeated $n = \log_2 N$ times, resulting in the computation of the desired discrete Fourier transform in *normal order*. At each stage of the *FFT* algorithm $N/2$ *simplified-butterfly* computations are performed. The computational flow graph for *simplified-butterfly* computation is illustrated in Figure 2.1. On the other hand, in the *basic-butterfly* scheme two complex multiplications and two additions/subtractions are performed on each *FFT* point. Fig. 2.2 shows clearly the redundant multiplication which is

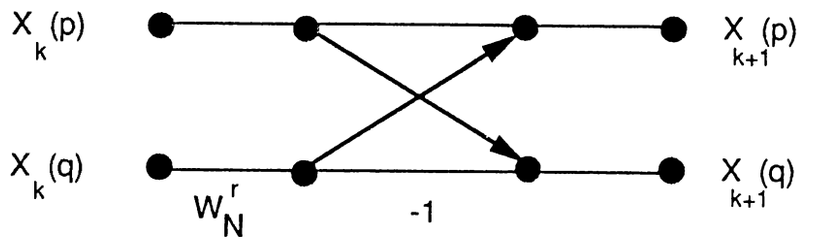

Figure 2.1. Simplified Butterfly.

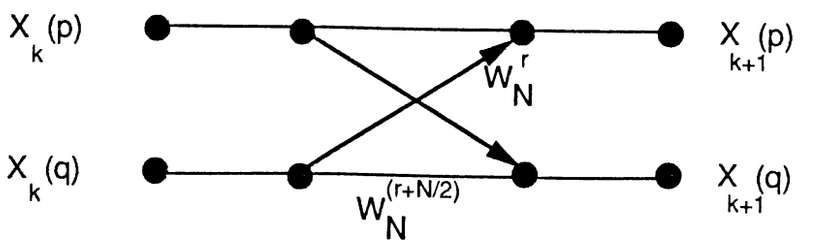

Figure 2.2. Basic Butterfly.

discarded in the *simplified-butterfly* scheme.

The *simplified-butterfly* computations required at the k^{th} stage of an N-point FFT algorithm is,

$$\begin{aligned} temp &= W_N^r \times X_k(q) \\ X_{k+1}(p) &= X_k(p) + temp \\ X_{k+1}(q) &= X_k(p) - temp \end{aligned} \tag{2.2}$$

where $q = p + 2^k$, $W_N^r = e^{-j(\frac{2\Pi}{N})r}$. That is, at the k^{th} stage, $N/2$ *simplified-butterfly* computations are performed on partially transformed pairs separated by 2^k. The pseudo-code for an N-point FFT algorithm is given in Prog. 2.1.

The *SEQFFTk* function shown in this program performs $N/2$ *simplified-butterfly* computations required at the k^{th} stage of the algorithm. The *SEQFFTk* function is invoked $n = \log_2 N$ times to compute the complete N-point *FFT*. In this program, N-complex inputs $x_0(i)$ $(i = 0, 1, \ldots, N-1)$ are assumed to be stored, in *bit-reversed* order, in one dimensional X-array of size N. Note that, after the initialization of input in the X-array, the computations are performed in place and the results are obtained in the X-array in *normal order* [5].

As is seen in Eq. 2.2 and Prog. 2.1 only one complex multiplication and two complex addition/subtraction operations are required in a butterfly computation. Since $N/2$ butterfly computations are performed at each stage, the *FFT* algorithm requires $(N/2)\log_2 N$ complex multiplications and $N \log_2 N$ complex additions and subtractions. Hence, the calculation of an N-point *FFT* requires $2N \log_2 N$ real multiplications and $3N \log_2 N$ real additions. Since the real multiplication and addition takes almost equal amount of time in most of the current arithmetic coprocessors (e.g., 80387[6]), the complexity of the sequential *FFT* can be expressed as,

$$T_{P1} = [5N \log_2 N] t_{calc} \tag{2.3}$$

where t_{calc} is the time taken by a floating point multiplication/addition.

The computations of the complex coefficients W_N^r are not involved in the given complexity analysis. In most of the real time applications (e.g., radar applications), N-point *FFT* is applied consecutively, for a fixed N, to N-point input data sets. Hence, the computation of the $Wfac$ coefficient values (twiddle factors) as they are needed, will be extremely inefficient. Hence, in general, $N/2$ coefficient values are computed once, as the value of the N is fixed, and stored in a table. These coefficients are then accessed by a simple table lookup procedure during successive *FFT* computations. Thus, the computational complexity of the coefficient calculations are negligible in such applications.

```
/* Input in bit-reversed order in X[0 ... N-1]   */
/* Output in normal order in X[0 ... N-1]       */
n := log2 N
for k :=0 to n − 1 do
    Call SEQFFTk (X, Wfac, N, k)
endfor
/* Performs N/2 Butterfly computations over the k^th bit. */
SEQFFTk (X,Wfac,N,k)
    for i :=0 to (N/2^(k+1)) − 1 do
        for j :=0 to 2^k − 1 do
            p := i × 2^(k+1) + j
            q := p + 2^k
            temp := Wfac × X(q)
            X(q) := X(p) − temp
            X(p) := X(p) + temp
        endfor
    endfor
end SEQFFTk
```

Program 2.1 : Sequential N-pt FFT Algorithm

2.3 Parallel FFT Algorithms

Butterfly computation of the *FFT* algorithm is very suitable for parallelization on multicomputers implementing the hypercube interconnection topology. The distribution of data and computations is straightforward for coarse *grain* parallelism whenever the number of *FFT* points, $N = 2^n$, is greater than or equal to the number of processors, $P = 2^d$, in the hypercube. The straightforward mapping can easily be achieved by using the decimal ordering of the processors in the hypercube. The first processor in the decimal ordering is assigned the first $M = N/P$ points of the X-array, the second processor will be assigned the next M-points and so on. Hence, successive processors in the decimal ordering will be assigned the successive slices of the X-array with each slice containing equal amount of, $M = N/P$, *FFT* points. The decomposition of a 16-point *FFT* data and computations on a 2-dimensional hypercube, with $M = 4$ *FFT* points assigned to each processor, is illustrated in Figure 2.3. In this scheme, each processor is responsible for carrying out the complete in place computations required for the *FFT*-points assigned to itself. As is seen in Figure 2.3, in

the first $(n-d)$ stages of the *FFT* algorithm butterfly computations are performed for pairs separated by $2^0, 2^1, \ldots, 2^{n-d-1}$. Hence, no communication is required between processors during the first $(n-d)$ stages since butterfly pairs separated by up to $M/2 = N/2P = 2^n/(2*2^d) = 2^{n-d-1}$ are assigned in groups to the same individual processors. In the last d stages of the FFT algorithm, butterfly pairs are separated by $2^{n-d}, 2^{n-d+1}, \ldots, 2^{n-1}$ where p and q points of each butterfly pair are assigned to different neighbor processors whose indices differ by $2^0, 2^1, \ldots, 2^{d-1}$ respectively. Hence, d-concurrent *exchange* communication steps are required in the last d stages of the parallel *FFT* algorithm where $d = \log_2 P$. A pseudo-code for the node program of the parallel *FFT* algorithm is given in Prog. 2.2.

The function *SEQFFTk* given in Prog. 2.1 performs the in place (in X-array) computations corresponding to the k^{th} stage of an N-point *FFT*. The first *for-loop* in Prog. 2.2 accumulates the summations over the first $(n-d)$ bits by performing butterfly computations for the local p, q pairs separated by $2^0, 2^1, 2^2, \ldots, 2^{n-d-1}$. Hence, the first *for-loop* involves no interprocessor communication. The second *for-loop* in Prog. 2.2 accumulates the summations over the last d-bits by performing butterfly computations for the p, q pairs separated by $2^{n-d}, 2^{n-d+1}, \ldots, 2^{n-1}$. Hence, an N-point data exchange is performed concurrently at each iteration of the second *for-loop* by issuing a send/receive message pair. As is seen in Prog. 2.2, this scheme introduces a storage overhead of size N/P per processor due to the local receive buffer *XRB*-array.

The straightforward mapping scheme described above achieves mapping the *FFT*-point groups that need mutual communication into nearest-neighbor processors of the hypercube. However, this scheme has two major drawbacks. First, in the given scheme, partially transformed N/P *FFT*-points have to be exchanged between pairs of processors at each stage of the last d stages. Second, perfect load balance is disturbed during the last d stages. These two drawbacks can explicitly be seen when Prog. 2.2 is analyzed. At each iteration of the second *for-loop*, one half of the processors hold only the updated values

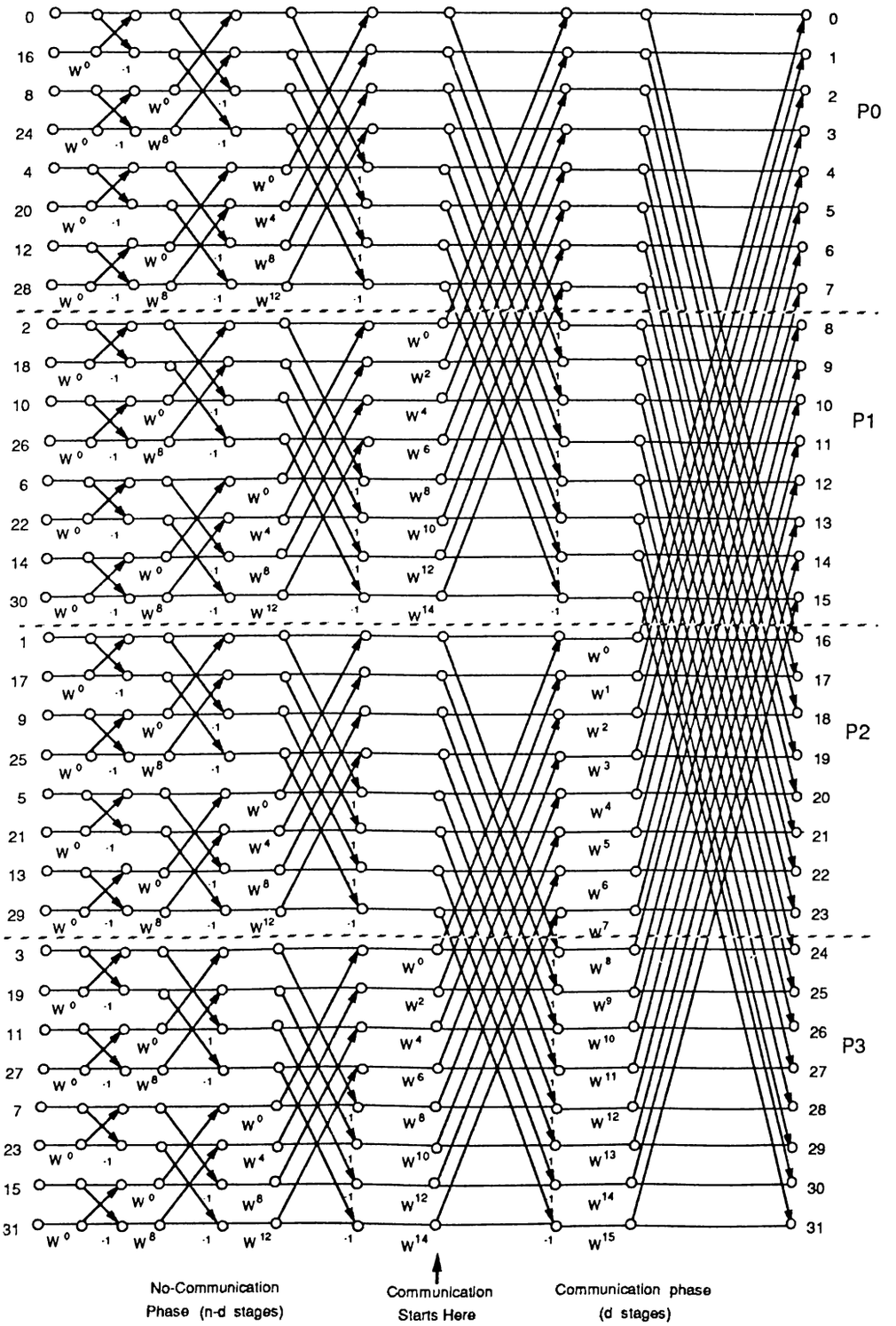

Figure 2.3. Computational flow graph for a 16-point FFT and its *static* mapping on an hypercube with 4-processors.

```
/* Computation over the first (n − d) bits: No communication   */
n := log2 N;  d := log2 P;  M := N/P;  m := log2 M;
for k :=0 to n − d − 1 do
    Call SEQFFTk (X,Wfac,M,k)
endfor
/* Computation over the next; hence the last d bits   */
/* d concurrent exchange steps   */
for k :=n − d to n − 1 do
    ℓ = k − (n − d);    dnode := mynode ⊕ 2^ℓ;
    if ((ℓ^th bit of mynode) = 1) then do
        for i :=0 to M − 1 do
            X(i) := Wfac × X(i)
        endfor
        csend from (X(i): i=0,1,2, ..., M-1) to dnode
        crecv into (XRB(i): i=0,1,2, ..., M-1) from dnode
        for i :=0 to M − 1 do
            X(i) := XRB(i) − X(i)
        endfor
    else
        csend from (X(i): i=0,1,2, ..., M-1) to dnode
        crecv into (XRB(i): i=0,1,2, ..., M-1) from dnode
        for i :=0 to M − 1 do
            X(i) := X(i) + XRB(i)
        endfor
        endif
endfor
```

Program 2.2 : Parallel N-pt FFT Algorithm on a hypercube with P processors

for the p-points and the other half hold only the updated values for q-points of the butterfly pairs. Since only the updated values of the q-points of the butterfly pairs have to be multiplied by the coefficients those processors holding N/P q-points performs N/P complex multiplications and the other processors waits idle for receiving the multiplication results from those processors. Hence, the parallel complexity of this scheme is,

$$T_{P2} = [\frac{5N}{P}\log_2\frac{N}{P}]t_{calc} + [\frac{8N}{P}\log_2 P]t_{calc} + [t_{su} + \frac{N}{P}t_{tr}]\log_2 P \quad (2.4)$$

Here, t_{su} represents the message startup overhead or the message latency and t_{tr} is the time taken for the transmission of a complex floating-point word ($2 * 4$ bytes) between two neighbor processors. Note that, the performance of the given scheme effectively degrades to the performance of the *basic-butterfly*

scheme (two complex multiplications per pair) during the last d stages. The communication time required for down-loading the input data and off-loading the results to/from the processors of the hypercube is not included in the above complexity model. However, each one of these operations can be performed in $\log_2 P$ communication steps.

2.3.1 Perfect Load Balance

Parallel *FFT* algorithms which achieve perfect-load balance are given in [7] for input in *bit-reversed* order and in [8] for input in *normal* order. Both of these two parallel algorithms exploit the *simplified-butterfly* scheme. The implementation details are not given explicitly in [8]. The parallel running time model given in [8] shows the volume of comunication as N/P complex *FFT* points per stage although it is indicated that only $N/2P$ complex *FFT* points are exchanged in a stage. The parallel program implemented in [7] performs two exchange communications per stage of the *FFT* algorithm. The algorithms given in [9] also achieve perfect load balance. However, these algorithms use *basic-butterfly* scheme for the *FFT* computations. In this subsection, we present a parallel *FFT* algorithm which achieves perfect load-balance for the *FFT* scheme using the *simplified-butterfly* with inputs in *bit-reversed* order. In the presented scheme, $N/2P$ complex *FFT* points are exchanged only once between processor pairs at each stage of the last d-stages.

The straightforward mapping scheme used in Prog. 2.2 already maintains perfect load balance during the first $(n-d)$ stages. Hence, this mapping is maintained during the first $(n-d)$ stages of the parallel algorithm proposed in this subsection. As is indicated earlier, one half of the processors hold only updated values for the p-points and the other half hold only the updated values for the q-points of the butterfly pairs during the last d-stages. This *static* mapping scheme is altered at the beginning of each stage of the last d stages. At the very beginning of each stage, each processor holding only updated values for the q-points ($N/2P$ *FFT* points) exchange one half of its q-points with the $N/2P$ p-points of its neighbor processor which holds all the p-points of its

butterfly pairs of that stage of the algorithm and vice versa. This exchange operation is not only the echange of data values to be used at that stage of the algorithm. In fact, processors effectively exchange the responsibility of the further *FFT* computations associated with those exchanged *FFT* points.

This *dynamic* mapping scheme is illustrated in Figure 2.4 for a 16-point *FFT* on a 4-processor hypercube. The pseude-code for the node-program of the proposed parallel *FFT* algoithm is given in Prog. 2.3. A C-like notation is used in Prog. 2.3 and throughout the chapter to represent the *for-loops*. The initial mapping scheme and the first *for-loop* in Prog. 2.3 is exactly similar to Prog. 2.2. It is the second *for-loop* where those two programs differ from each other. As is seen in Figure 2.4 and Prog. 2.3, each processor exchange either the first half or the second half of its local X-array in place by simply checking the ℓ^{th} bit value of its processor index where ℓ denotes the channel over which the exchange operation is to be performed on that stage. Due to the *dynamic* mapping scheme, each processor performs *simplified butterfly* computations on local p and q pairs separated by $2^{m-1} = N/2P$ after the exchange operations at each stage of the last d stages. Hence, in this scheme, each processor holds equal number of p and q points ($N/2P$ p-points and $N/2P$ q-points) after the exchange operation. Each processor performs equal number of $(N/2P)$ complex multiplication thus achieving a perfect load balance. Furthermore, the volume of communication at each exchange communication step is reduced by a factor of two (from N/P to $N/2P$ complex floating-point words). Hence, the parallel complexity of the proposed scheme is,

$$T_{P3} = [\frac{5N}{P}\log_2 N]t_{calc} + [t_{su} + \alpha\frac{N}{2P}t_{tr}]\log_2 P + \frac{N}{2P}t_{copy} \qquad (2.5)$$

In the complexity model given above, two concurrent *send* communication operations (for an *exchange* operation) between a pair of processor are assumed to be overlapped completely. The *startup* overheads (t_{su}) for the mutual *send* operations are in fact overlapped completely. The overlap of mutual data *transmission* ($\frac{N}{2P}t_{tr}$) between a pair of processor is feasible only when two physical

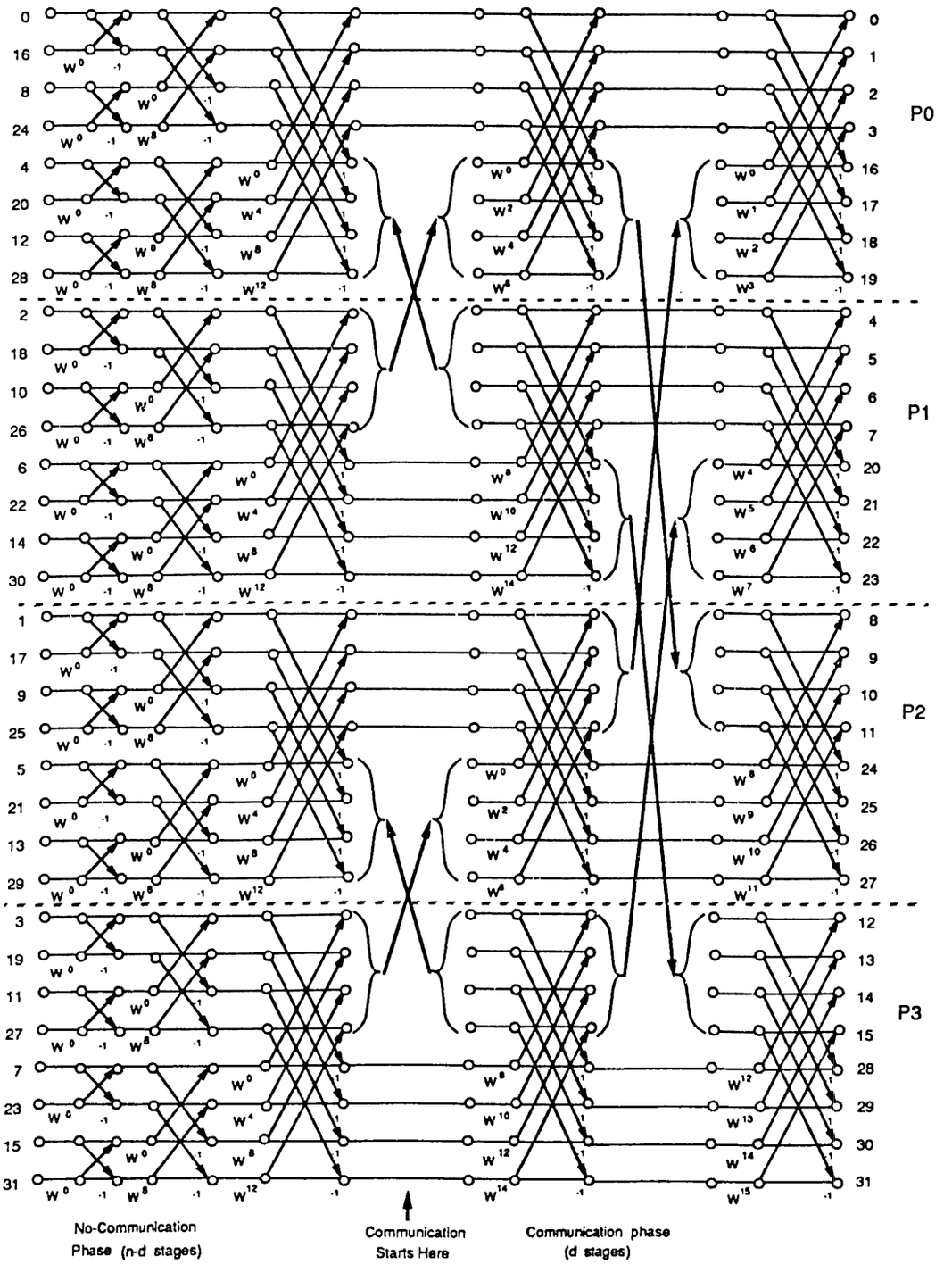

Figure 2.4. Dynamic mapping of 16-point FFT data and computations on a hypercube with 4-processors.

links are present between neighbor processors as in iPSC/2. However, the internal hardware architecture of an individual iPSC/2 processor is such that internal bus conflicts occur due to the outgoing and incoming long messages during an *exchange* operation. Hence, a complete overlap cannot be achieved in iPSC/2 during the mutual data *transmission* phase of the *exchange* operation. The performance of the exchange operation can be modeled as $[t_{su} + \alpha \times \frac{N}{2P} t_{tr}]$ in iPSC/2, where α is measured to be $1.3 \leq \alpha \leq 1.6$ in [10]. The variation of α with respect to concurrent incoming and outgoing message length is given in Figure 2.5. Hence, this α parameter should be inserted as a coefficient to the $\frac{N}{2P} t_{tr}$ term in Eq. 2.5 to give the running time model of the parallel algorithm on iPSC/2. Note that, in Prog. 2.3, each processor issues *synchronous receive* just after the *synchronous send* operation. Due to the perfect load balance, communicating processor pairs perform the *synchronous send* operations concurrently. A *synchronous send* operation returns the control back to node program only after the outgoing message leaves the indicated send buffer XSB-array. Whenever an incoming message begins to arrive to a destination processor, it does not find a pending receive, hence it is copied to a temporary system buffer by the node operating system NX. Later, whenever the receive operation is issued by the node program, that message is copied from the temporary system buffer to the indicated receive buffer X-array. Hence, late issue of the synchronous receive message introduces a block copy overhead for $N/2P$ complex floating-point words. The last term in Eq. 2.5 accounts for this receive overhead where t_{copy} represents the time taken for the copy of a single complex floating-point word from the system buffer to the indicated receive buffer X-array. Note that, such a receive overhead is not included in the parallel time complexity model given in Eq. 2.4 for Prog. 2.2. In Prog. 2.2, due to the lack of load balance, communicating processor pairs do not initiate send operations concurrently. The model in Eq. 2.4 is given for the bottleneck processors which stay idle, waiting for the multiplication results from their neighbor processors. These processors always issue an early synchronous receive. Due to a pending receive for an incoming message, the incoming message is directly copied into the indicated receive buffer XRB-array.

Eq. 2.5 reveals the fact that the proposed scheme achieves a perfect load

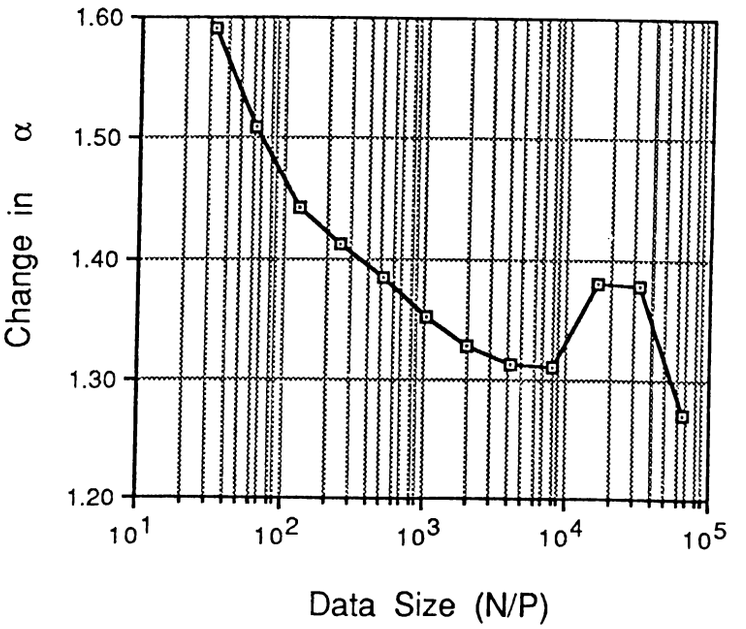

Figure 2.5. Variation in α with respect to $N/2P$.

/* Computation over the first $(n-d-1)$ bits: No communication */
n := $\log_2 N$; d := $\log_2 P$; M := N/P; m := $\log_2 M$;
for k :=0 **to** $n-d-1$ **do**
  Call *SEQFFTk* (X, Wfac, M, k)
endfor
/* Computation over the next d bits: d concurrent exchange steps */
for k :=$n-d$ **to** $n-1$ **do**
  $\ell := k-(n-d)$; dnode := mynode $\oplus\ 2^{\ell}$;
  if ($(\ell^{th}$ bit of mynode) = 1) **then do**
    csend from (X(p): p=0,1,2, ..., $M/2-1$) to *dnode*
    crecv into (X(q): q=0,1,2, ..., $M/2-1$) from *dnode*
  else
    csend from (X(q): q=$M/2, M/2-1, M/2-2 \ldots, M-1$) to *dnode*
    crecv into (X(p): p=$M/2, M/2-1, M/2-2 \ldots, M-1$) from *dnode*
  endif
  for (p :=0, q :=$M/2$; p < $M/2$; p++, q++) **do**
    temp := Wfac × X(q)
    X(q) := X(p) − temp
    X(p) := X(p) + temp
  endfor
endfor

Program 2.3 : Parallel N-pt FFT Algorithm with Perfect Load-Balance on a hypercube with P processors

balance. As is seen in Prog. 2.3, this scheme requires no extra storage for *send/receive* buffers since *send/receive* operations are performed in place to or from the local X-array. As is seen in Figure 2.4, the output results are slightly scrambled (in $N/2P$ blocks) in this scheme due to the *dynamic* mapping scheme proposed. However, the off-loading of the results from the processors of the hypercube can also be performed in $\log_2 P$ communication steps without increasing the volume of communication.

2.3.2 Overlapping Communication with Computation

There are strong data dependencies in the *FFT* algorithm. The update of each *FFT* point requires communication in the last d-stages of the algorithm. Hence, as is also indicated in [9, 10], the *FFT* algorithm differs from local spatially decomposed problems such as Finite Difference and Finite Element problems. In such problems, communications associated with the boundary points can easily be overlapped with the updating of interior data points[9]. Thus, communication and computation in the *FFT* algorithm cannot be overlapped easily.

Zhu has proposed a scheme for overlapping communication with the computation of the complex coefficients (complex exponentiations) in [8]. However, as is discussed earlier, computation of the coefficients as they are needed is not an efficient scheme compared to the table lookup scheme. Walker has proposed a scheme in [9] for overlapping communication and computation for the *FFT* algorithm using the *basic-butterfly* scheme which requires two complex multiplication per butterfly pair. In this section, we propose two schemes which overlap the communication with one fifth of the computations involved in a stage of the *FFT* algorithm which uses the *simplified-butterfly* and the table lookup scheme.

Scheme 1: Asynchronous Send and Synchronous Receive

The pseudo-code for the node program of the parallel *FFT* algorithm which overlaps communication and computation is given in Prog. 2.4. The initial *static* mapping for the first $(n - d)$ stages and the *dynamic* mapping for the last d-stages is similar to the scheme given in Prog. 2.3. However, as is seen in Prog. 2.4, the first *for-loop* iterates only $(n - d - 1)$ times which is one less than the iteration count in Prog. 2.3. Then at each iteration (computation stage) of the second *for-loop* each processor pipelines the *send* portion of the exchange operation required at the following stage in the current stage. Each processor classifies its computational task at each stage into two categories: those updates to be *sent* to the destination processor in the following stage and other updates to be kept as local in the following stage. Then, each processor first performs the computations associated with those points required by the destination processor in the next stage. Hence, each processor first performs $N/2P$ complex multiplications associated with its local $N/2P$ q-points. Then, each processor updates either the values of its local p-points or q-points simply by checking the $(\ell + 1)^{th}$ bit value of its processor index. Here, $\ell + 1$ denotes the channel over which the exchange operation required in the next stage. Upon completion of these $N/2P$ updates, each processor issues an *asynchronous* (non-blocking) *send* to initiate the transmission of the updated $N/2P$ *FFT*-point values to the destination processor. After initiating the *send* operation, each processor completes the computation associated with that stage by updating other half of its local *FFT*-points that will be kept local in the following stage. Upon completion of the second type updates each processor issues a *synchronous receive* to complete the already initiated exchange operation.

As is seen in Prog. 2.4, the first inner *for-loop* of the second outer *for-loop* performs the updates to be send immediately to the destination processor. Each processor performs these updates into a send buffer (*XSB* array) of size $N/2P$ since it does not need these updates in further *FFT* computations. However, receive portion of the exchange operations can be done in place into the local X-array since *synchronous receive* is issued after the completion of the overall computations associated with that stage. Hence, the proposed scheme

```
/* Computation over the first (n − d − 1) bits: No communication   */
n := log2 N;   d := log2 P;   M := N/P;   m := log2 M;
for k :=0 to n − d − 2 do
    Call SEQFFTk (X, Wfac, M, k)
endfor
/* Computation over the next d bits: d-concurrent exchange comm. steps  */
for k :=n − d − 1 to n − 2 do
    ℓ := k − (n − d);    dnode := mynode ⊕ 2^(ℓ+1);
    if ((ℓ + 1)^th bit of mynode) = 1) then do
       for (p :=0, q :=M/2; p < M/2; p++, q++) do
           X(q) := Wfac × X(q)
           XSB(p) := X(p) + X(q)
       endfor
       isend from (XSB(p): p=0, 1, 2, ..., M/2 − 1) to dnode
       for (p :=0, q :=M/2; p < M/2; p++, q++) do
           X(q) := X(p) − X(q)
       endfor
       crecv into (X(p): p=0, 1, 2, ..., M/2 − 1) from dnode
    else
       for (p :=0, q :=M/2; p < M/2; p++, q++) do
           X(q) := Wfac × X(q)
           XSB(p) := X(p) − X(q)
       endfor
       isend from (XSB(q): q=0, 1, 2, ..., M/2 − 1) to dnode
       for (p :=0, q :=M/2; p < M/2; p++, q++) do
           X(p) := X(p) + X(q)
       endfor
       crecv into (X(q): q=M/2, M/2 + 1, M/2 + 2..., M − 1) from dnode
    endif
    msgwait on isend
endfor
/* Perform M/2 Butterfly computations over the local (m − 1)^th bit   */
Call SEQFFTk (X, Wfac, M, m − 1)
```

Program 2.4 : Overlapped Parallel N-pt FFT Algorithm with Perfect Load-Balance on a hypercube with P processors Scheme #1

introduces a storage overhead of size $N/2P$ per processor due to the local send buffer *XSB* array. Note that, the size of the local X-array is N/P. The only computational overhead is the loop overhead since two *for-loops* are required instead of one. The number of floating-point computations is exactly equal to the number of computations required in Prog. 2.3. As is seen in Prog. 2.4, $N/2P$ complex additions/subtractions shown in the second inner *for-loop* of the second outer *for-loop* are overlapped with communication. Hence, communication is overlapped with one fifth of the computations involved in a stage. Thus, the parallel complexity of the proposed algorithm is

$$\begin{aligned} T_{P4} &= [\frac{5N}{P}\log_2\frac{N}{P}]t_{calc} + [\frac{4N}{P}\log_2 P]t_{calc} + \\ &\quad [Max\{\frac{N}{P}t_{calc},\ (t_{su} + \alpha\frac{N}{2P}t_{tr})\}]\log_2 P + \frac{N}{2P}t_{copy} \end{aligned} \tag{2.6}$$

Hence, for sufficiently large N/P, where,

$$\frac{N}{P}t_{calc} \geq t_{su} + \frac{N}{2P}t_{tr} \tag{2.7}$$

complete overlap of communication can be achieved.

Scheme 2: Asynchronous Send and Asynchronous Receive

As is seen in Prog. 2.4, only the *send* portion of the communication is overlapped with one fifth of the computation. Note that, in Prog. 2.4, each processor issues *synchronous receive* after initiating *asynchronous send* operation and performing $N/2P$ complex addition/subtraction operations. Due to the perfect load balance, communicating processor pairs initiate the *asynchronous send* operations concurrently. Hence, there is no pending receive in a destination processor for an incoming message. The last term in Eq. 2.6 accounts for the receive overhead due to the late issue of the *synchronous receive* operation. If, however, there is a pending *asynchronous receive* for an incoming message, then it is directly copied into the indicated receive buffer. Hence, it is also possible to overlap the receive portion by issuing an early *asynchronous receive*. In this section, we present a second scheme which overlaps both *send/receive* portions of the communication with computation using the *simplified-butterfly* and the table lookup methods described before. The pseudo-code for the node

program of the parallel *FFT* algorithm which overlaps both *send/receive* operations with computation is given in Prog. 2.5. Because the whole algorithm is lengthy, we only give the d-concurrent exchange phase of Prog. 2.5. The remaining part is similar to Prog. 2.4.

As is discussed in the previous section, the static mapping for the first $n-d$ stages and the *dynamic* mapping for the last d stages are similar to the algorithm given in Prog. 2.4. The main difference between them is; in Prog. 2.5 we issue an *asynchronous* (non-blocking) *receive* at the beginning of each iteration of the d concurrent exchange phase in order to initiate the *receive* operation before any *asynchronous* (non-blocking) *send* is issued. This scheme outperforms Prog. 2.4, since in this case, we also overlap the *receive* operation as well as *send* operation. On the other hand, the overhead introduced to the original program (i.e., Prog. 2.3) is only the *for-loop* overhead in Prog. 2.4. The only disadvantage of this scheme is that, it brings an additional N/P memory overhead compared to Prog. 2.4. In this scheme, we use an X-array of size $2N/P$ compared to N/P in Prog. 2.4. The second half of the X-array is used as a *receive* buffer of length N/P. The first half of the *receive* buffer is named as $XRB\text{-}ODD$ and the second half as $XRB\text{-}EVEN$. According to the cycle count incoming message is received either into $XRB\text{-}ODD$ or $XRB\text{-}EVEN$. This switching receive buffer scheme is chosen to avoid the contamination of the message received in the previous cycle by the incoming message expected in the current cycle.

At the beginning of each iteration during the d-concurrent exchange phase, each processor issues an *asynchronous* receive operation. According to the value of the cycle count, odd or even, the receive operation is initiated to the respective buffer (i.e., $XRB\text{-}ODD$ or $XRB\text{-}EVEN$). Processors use $XRB\text{-}ODD$ and $XRB\text{-}EVEN$ buffers one after other in an alternating fashion. If a processor has used $XRB\text{-}ODD$ in the previous cycle then it should use $XRB\text{-}EVEN$ in the current cycle and vice versa. After initiating the receive operation, each processor calculates the address of p and q pairs. If in the previous cycle the *send* operation was for p-points then the address of p-points

```
/* Computation over the next d bits: d-concurrent exchange comm. steps */
cycle := 0;
for k :=n − d − 1 to n − 2 do
    ℓ := k − (n − d);   dnode := mynode ⊕ 2^(ℓ+1);   cycle := cycle + 1;
    if (cycle = odd) then do
      irecv into (X(p): p=M, M + 1, M + 2, ..., 3M/2 − 1)
    else
      irecv into (X(q): q=3M/2, 3M/2 + 1, 3M/2 + 2, ..., 2M − 1)
    endif
    Calculate pstart and qstart
    qptr := qstart;   pptr := pstart;
    if ((ℓ + 1)^th bit of mynode) = 1) then do
       for (p :=0, q :=M/2; p< M/2; p++, q++, pptr++, qptr++) do
           X(q) := Wfac × X(qptr)
           XSB(p) := X(pptr) + X(q)
       endfor
       isend from (XSB(p): p=0, 1, 2, ..., M/2 − 1) to dnode
       for (p :=0, q :=M/2, pptr := pstart; p< M/2; p++, q++, pptr++) do
           X(q) := X(pptr) − X(q)
       endfor
    else
       for (p :=0, q :=M/2; p< M/2; p++, q++, pptr++, qptr++) do
           X(q) := Wfac × X(qptr)
           XSB(p) := X(pptr) − X(q)
       endfor
       isend from (XSB(q): q=0, 1, 2, ..., M/2 − 1) to dnode
       for (p :=0, q :=M/2, pptr := pstart; p< M/2; p++, q++, pptr++) do
          X(p) := X(pptr) + X(q)
       endfor
    endif
    msgwait on isend
    msgwait on irecv
endfor
```

Program 2.5 : Overlapped Parallel N-pt FFT Algorithm with Perfect Load-Balance on a hypercube with P processors Scheme #2 (*d-concurrent exchange phase*)

in this cycle is either $XRB\text{-}ODD$ or $XRB\text{-}EVEN$ and the address of q-points is in place. On the other hand, if the *send* operation was for q-points then the address of q-points is either $XRB\text{-}ODD$ or $XRB\text{-}EVEN$ and the address of p-points is in place too. This address calculation scheme makes sure that every processor finds p and q pairs correctly in X-array.

During the d concurrent exchange phase, each processor classifies its computational task at each iteration similar to the one in Prog. 2.4. Also the calculations done inside the second *for-loop* are the same as in Prog. 2.4. The only difference is at the end of the calculations; each processor issues a *msgwait* operation on already started *asynchronous Send/Receive* operations in order to complete the exchange operations.

In order to be complete, a *prelude* and a *postlude* section is included before and after the d concurrent exchange phase. The *prelude* section, organizes the data according to the d concurrent exchange phase. The *postlude* section, is similar to $SEQFFTk$. The only difference is that, in the *postlude* section p and q pairs can be in $XRB\text{-}ODD/XRB\text{-}EVEN$. The postlude code handles it.

As is seen in Prog. 2.5, the size of the local X-array is $2N/P$. The second half of it is used as two consecutive receive buffers (of size $N/2P$) named as $XRB\text{-}ODD$ and $XRB\text{-}EVEN$. The size of XSB-array is $N/2P$ as in the case of Prog. 2.4. Hence this scheme introduces an extra storage overhead of N/P for the receive buffer, compared to Prog. 2.4. On the other hand, the introduced computational overhead is as same as in Prog. 2.4, only an extra *for-loop*. The number of floating point computations associated in each stage is exactly equal to the number of computations required in Prog. 2.3 and Prog. 2.4. Thus the parallel complexity of the proposed algorithm can be written as:

$$\begin{aligned} T_{P5} &= [\frac{5N}{P}\log_2\frac{N}{P}]t_{calc} + [\frac{4N}{P}\log_2 P]t_{calc} + \\ &\quad [Max\{\frac{N}{P}t_{calc},\ (t_{su} + \alpha\frac{N}{2P}t_{tr})\}]\log_2 P \end{aligned} \tag{2.8}$$

As is seen in Eq. 2.8, receive overhead is avoided by the early issue of the

asynchronous receive. It should be noted here that, the parameter α in Eq. 2.6 and Eq. 2.8 is expected to be slightly greater than the α used in Eq. 2.5 for iPSC/2 architecture. The reason is the expected increase in the local bus conflicts due to the local FFT computations overlapped with the outgoing and incoming messages.

2.4 Experimental Results

All programs presented in this chapter have been coded in C language and run on an 32-node iPSC/2 hypercube multicomputer for various $N = 2^n$ data sizes, $64 \leq N \leq 64K$. Figure 2.6 and Figure 2.7 displays the variation of percent overlap achieved in Prog. 2.4 and Prog. 2.5 respectively. Total communication time and overlapped communication times in Prog. 2.4 and Prog. 2.5 are computed by running Prog. 2.3 without invoking *csend* and *crecv* communication routines and subtracting these timings from the original execution timings of Prog. 2.3, Prog. 2.4 and Prog. 2.5 respectively. Percent overlap is then computed by dividing overlapped communication times by total communication times. As is seen in Figure 2.6 and Figure 2.7, percent overlap increases with increasing data size as expected. Note that, *for-loop* overhead is also included in these timings.

For small data sizes, the amount of computation is not large enough to achieve complete overlap of the communication (Eq. 2.7). Hence, negative percent overlap values are obtained for small data sizes. The oscillations seen in Figure 2.6 and Figure 2.7 for small data sizes are due to the resolution of the system clock used for timings. As is also seen in Figure 2.6 and Figure 2.7, percent overlap begins to decrease slightly after reaching a maximum value at large data sizes ($16K \leq N/2P \leq 32K$).

This decrease is closely related to the change in variation of α for those large data sizes. A maximum overlap of % 23 and % 54 are obtained in Prog. 2.4 and Prog. 2.5 respectively in spite of the *for-loop* overhead. Also, as is indicated

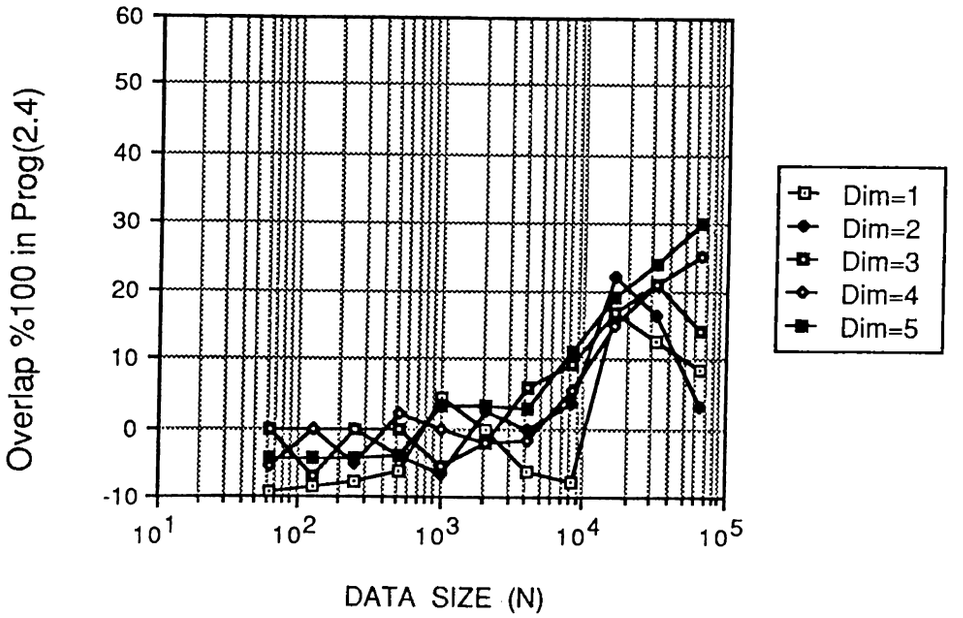

Figure 2.6. Percent Overlap curve for Program (2.4) compared to Program(2.3).

in [6], a complete overlap cannot be achieved due to the internal architecture of an individual iPSC/2 processor. Figure 2.8, illustrates the variation of percent performance improvement of Prog. 2.3 compared to Prog. 2.2 during the d exchange communication phases. As is seen in Figure 2.8, Prog. 2.3 outperforms Prog. 2.2 as expected since it achieves perfect load balance and reduces volume of communication by a factor of two compared to Prog. 2.2. As is also seen in Figure 2.8, percent performance improvement of Prog. 2.3 compared to Prog. 2.2 increases with increasing data size.

Figure 2.9 and Figure 2.10 displays the variation of percent performance improvement of Prog. 2.4 and Prog. 2.5 compared to Prog. 2.3 during the d exchange communication phase. As is seen in Figure 2.9 and Figure 2.10, the variation of percent performance improvement is very similar to the variation of overlap curves in Figure 2.6 and Figure 2.7 as expected. As is seen in Figure 2.9, Prog. 2.4 gives better performance results compared to Prog. 2.3 for $N/P \geq 8K$ (e.g. N=16K and P=2). On the other hand, as is seen in Figure 2.10, Prog. 2.5 gives better performance results compared to Prog. 2.3 for

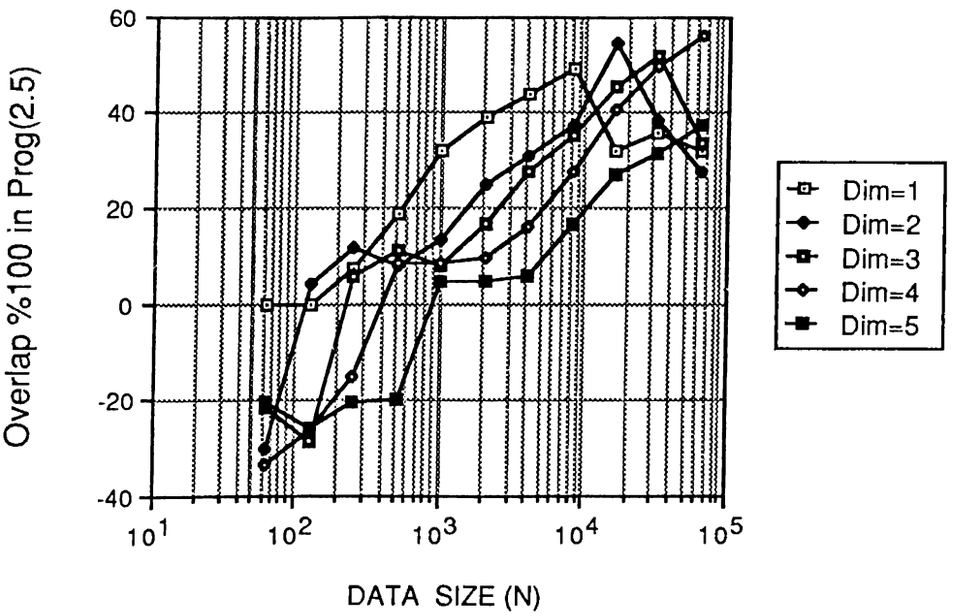

Figure 2.7. Percent Overlap curve for Program (2.5) compared to Program (2.3).

$N/P \geq 32$ (e.g. N=256 and P=4). Comparing Prog. 2.4 and Prog. 2.5, obviously Prog. 2.5 outperforms Prog. 2.4 for all N. This result is expected since the introduced overheads both in Prog. 2.4 & Prog. 2.5 are equal compared to Prog. 2.3. Furthermore in Prog. 2.4 only *send* operation is overlapped while in Prog. 2.5 both *send* and *receive* operations are overlapped.

The main reason why Prog. 2.4 and Prog. 2.5 doesn't give better performance results compared to Prog. 2.3 for $N/P < 8K$ and $N/P < 32$ is due to the extra overhead which is introduced with the second *for-loop*. This *for-loop* overhead in the algorithm cannot be assumed to be negligible and must be taken into account since it is equal to $3\mu sec/FFT$ point (experimentally calculated value). While on the other hand an empty *for-loop* overhead is equal to $1.2\mu sec/FFT$ point. The reason why this *for-loop* introduces a considerable amount of overhead (i.e., more than an empty *for-loop* is heavily dependent on the structure of 80387 and the compiler used in the iPSC/2 system.

Figure 2.11 and Figure 2.12 shows speed-up and efficiency curves for Prog. 2.5.

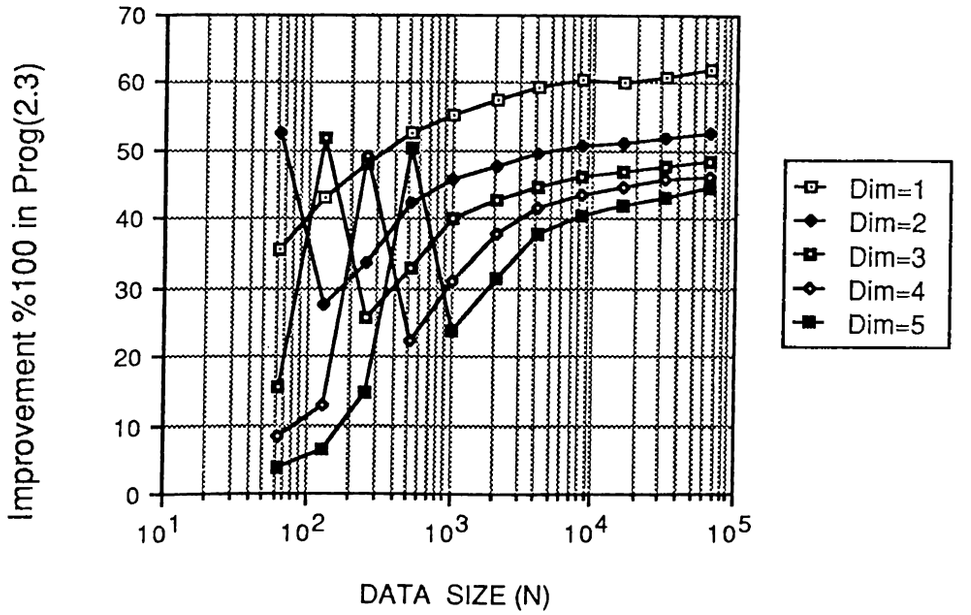

Figure 2.8. Percent Improvement of Program (2.3) over Program (2.2) during the last d-stages.

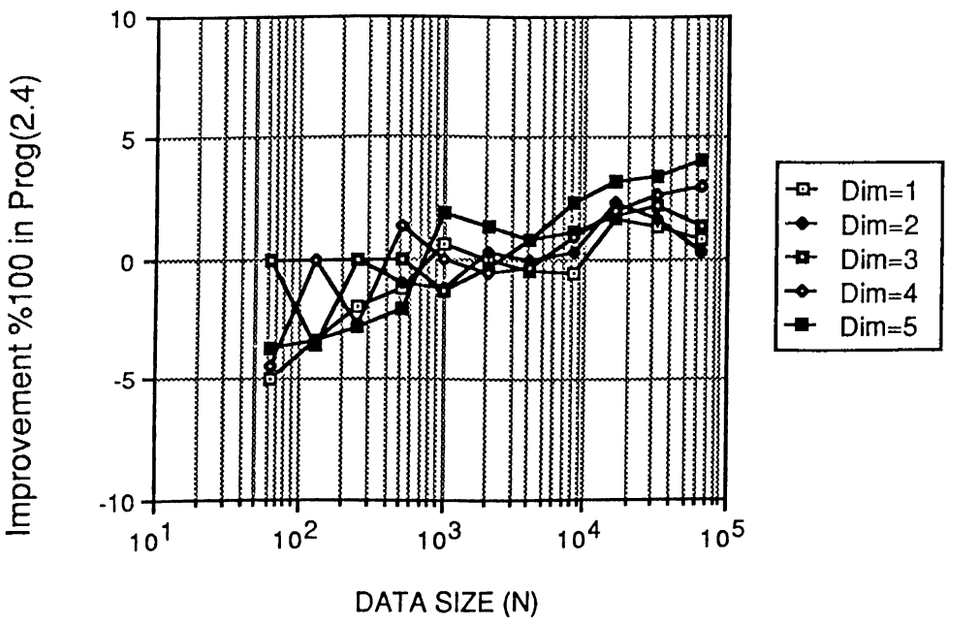

Figure 2.9. Percent Improvement of Program (2.4) over Program (2.3) during the last d-stages.

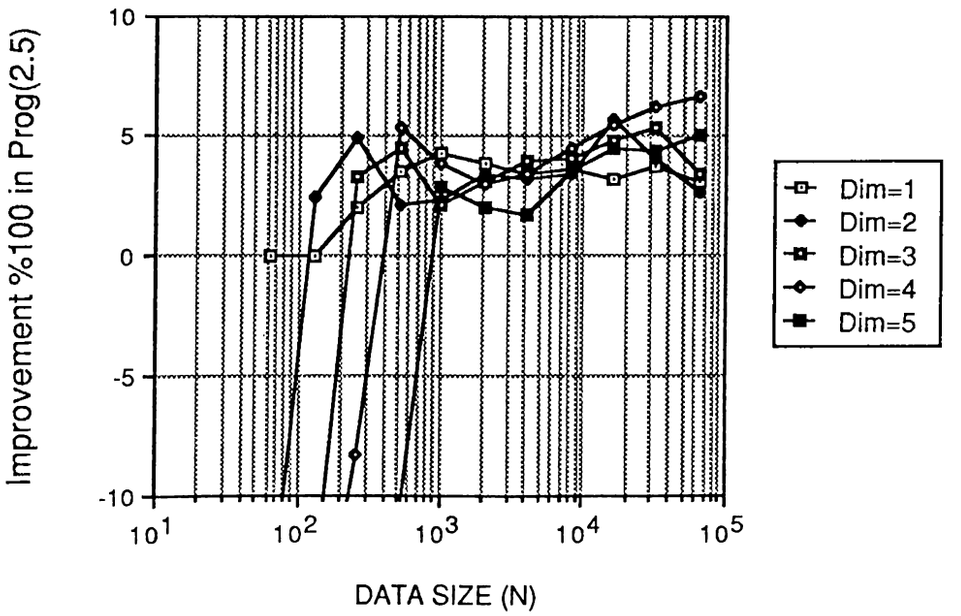

Figure 2.10. Percent Improvement of Program (2.5) over Program (2.3) during the last d-stages.

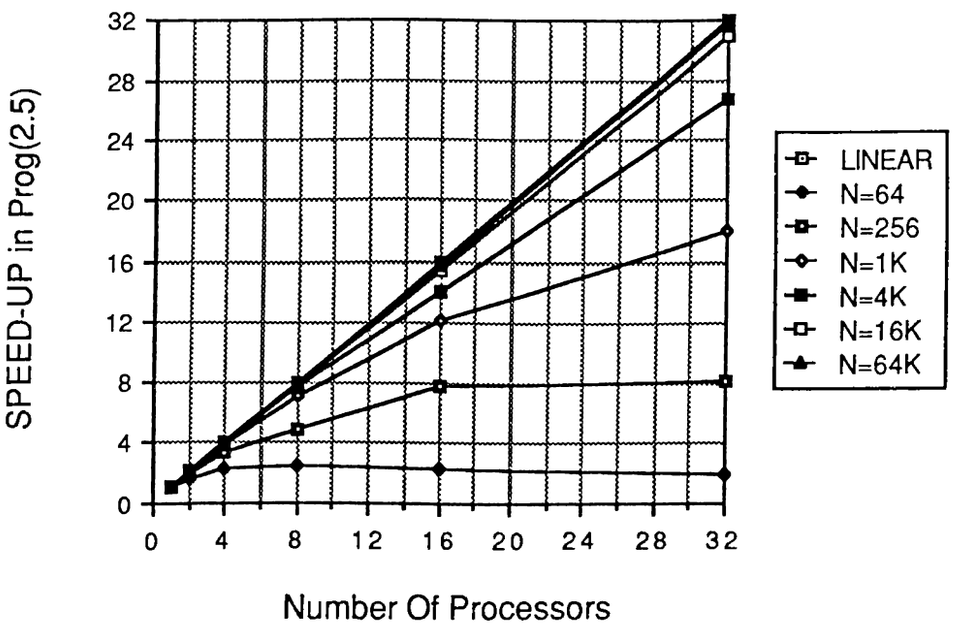

Figure 2.11. Speedup curve for Program (2.5).

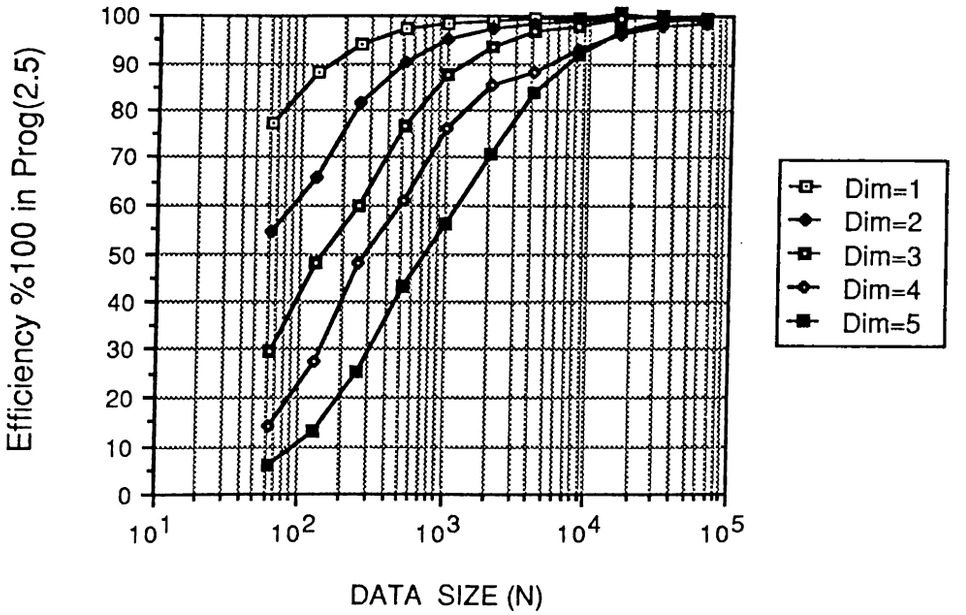

Figure 2.12. Efficiency curve for Program (2.5).

As is seen in Figure 2.11, almost linear speed-up is acieved for $N > 1K$. As is seen, in Figure 2.12, efficiency remains over %90 when $N/P \geq 256$ FFT points mapped to an individual processor of the hypercube.

2.5 Conclusion

In this chapter we have discussed three efficient parallel FFT algorithms for coarse-grain, distributed-memory, message-passing multiprocessors implementing the hypercube interconnection topology. The first proposed parallel FFT algorithm, Prog. 2.3 achieves both perfect load balance, reduces the volume of communication by a factor of two and doesn't introduce any memory overhead. The second proposed parallel FFT algorithm, Prog. 2.4 overlaps one fifth of the computation with only the send portion of communication using *asynchronous send* and *synchronous receive*. This method introduces an extra *send* buffer of length $N/2P$. The third proposed parallel FFT algorithm, Prog. 2.5 overlaps one fifth of the computation with both send and receive portions of communication using *asynchronous send/receive*. This method further introduces an extra *receive* buffer of length N/P.

The overall improvement in Prog. 2.4 & Prog. 2.5 according to Prog. 2.3 is rather small in our case (i.e., for a maximum cube dimension of 3). The reason is, d concurrent exchange phase of the FFT algorithm takes very small amount of time in the overall computation. Also a complete overlap cannot be achieved due to the internal architecture of an individual iPSC/2 processor [6]. Obviously algorithms perform better on higher cube dimensions. Also for different architectures which can achieve complete overlap, the performance index of Prog. 2.4 and Prog. 2.5 will be much higher.

3. The Fast Hartley Transform

3.1 Introduction

The Fourier transform allows signals to be treated in the complex frequency domain. If the signals in the time domain are real, then the Fourier transform contains redundancy. The Fast Hartley Transform is an alternative computation to the complex Fast Fourier Transform when the input signal to be transformed consists of real numbers only. The Fast Hartley Transform is as fast as or faster than the Fast Fourier Transform and still serves for all needs in the digital signal processing world where Fourier transform is presently applied.

Recalling the Discrete Fourier Transform for an input sequence $f()$ is,

$$F(k) = \sum_{i=0}^{N-1} (f(i)[Cos(2\pi ki/N) - jSin(2\pi ki/N)]) \tag{3.1}$$

where (k=0,1,..., N-1) and N is the length of input sequence. As one can see, Discrete Fourier Transform as well as its fast transform includes complex arithmetics. For a more efficient, simple and faster transformation, the Hartley transform was developed [11]. Its discrete form is ,

$$H(k) = \sum_{i=0}^{N-1} (h(i)[Cos(2\pi ki/N) + Sin(2\pi ki/N)]) \tag{3.2}$$

where the input sequence $h()$ is constrained to real numbers only.

Hartley transform does not neccesitate any complex arithmetics. This important feature of the Hartley transform increases the performance of DHT by

a factor of two, while decreasing the memory requirements again by a factor of two at the same time. Both of the discrete transformations have a computation time which is proportional to N^2. Fast Fourier Transform reduces this time to $N \log_2 N$ [1]. Similar methods can also be applied to the Hartley transform and one can easily obtain a fast transformation for Hartley called Fast Hartley Transform or shortly FHT [12, 13]. There are several Fast Hartley Transform algorithms named as Radix-2 Decimation-in-Time FHT, Radix-2 Decimation-in-Frequency FHT, Radix-4 FHT, Split Radix FHT, Recursive FHT and Vector FHT [14, 15, 16, 17]. One can also calculate FHT through FFT or vice versa [14].

The Fast Hartley Transform presented in Section 3.2 is a decimation-in-time, radix-2 algorithm as presented in [15]. In Section 3.3, static mapping and parallelization of the choosen FHT scheme is discussed. Section 3.3.1 presents the proposed restructuring of the FHT algorithm for an efficient parallelization. The dynamic mapping scheme proposed for the restructured FHT algorithm is also presented in Section 3.3.1. Section 3.4 presents the experimental results and performance evaluation of the proposed algorithms on Intel's iPSC/2 hypercube multicomputer.

3.2 Sequential FHT Algorithm

Computational steps for a 32-point radix-2, decimation-in-time FHT algorithm is illustrated in Fig. 3.1. This tabular representation is proposed in [17]. The input in this scheme is N-real numbers in *bit-reversed* order. The output is N-real numbers in *normal* order. The $C(i)$ and $S(i)$ factors Fig. 3.1 represent $Cos(2\pi i/N)$ and $Sin(2\pi i/N)$ respectively. As is seen in Fig. 3.1, each level of FHT algorithm takes a set of N real numbers and transforms them into another set of N real numbers. This process is repeated $n = \log_2 N$ times, resulting in the *in-place* computation of the desired *Discrete Hartley Transform* in *normal* order. However the tabular representation is not sufficient for a detailed analysis of the computational interdependencies which is crucial for an efficient parallel algorithm design. In this work, computational

flow graph for the FHT algorithm is derived in order to explore the computational interdependencies.

A close examination of Fig. 3.1, reveals that FHT computations at each level resembles basic FFT butterfly computations. The FHT computations at the first level ($\ell = 0$) are similar to the first level of the FFT computations. That is, first level of the FHT algorithm consists of two point FFT *butterfly* type of computations. In the remaining levels ($1 \leq \ell \leq n-1$) FHT algorithm consists of 4-point butterfly computations. However, in the case of FHT, there are two types of butterfly computations. These two distinct types of butterfly computations will be referred here as *type*-1 and *type*-2 *basic butterfly* computations. Fig. 3.2 and Fig. 3.3 illustrates the computational flow graph for *type*-1 and *type*-2 *basic* FHT *butterflies* at the ℓ^{th} level ($1 \leq \ell \leq n-1$) of an N-point FHT algorithm respectively. In these figures, $C(i)$ and $S(i)$ factors refer to $Cos(2\pi i/N)$ and $Sin(2\pi i/N)$ respectively. Each type of FHT butterfly computations are identified by an ordered 4-tuplet $\{p, r, q, s\}$. Note that, both types of basic FHT butterflies consist of two stages.

In the first stage of a *type*-1 *basic* FHT butterfly, (q, s) pair is involved in two butterfly type of computations to generate 4 intermediate results. Each butterfly computation involves the multiplication of q and s points by Cos/Sin and Sin/Cos factor pairs respectively and pairwise addition of these four multiplication results. Hence, the first stage of a *type*-1 *basic* FHT *butterfly* involves 8 multiplications and 4 additions. In the first stage of a *type*-2 *basic* FHT *butterfly*, both q and s points are multiplied by $Cos + Sin$ (CS) factor pairs to generate 4 intermediate results. Hence, the first stage of *type*-2 *basic* FHT *butterfly* involves only 2 multiplications. In the second stages of both *type*-1 and *type*-2 *basic* FHT *butterfly* computations, these 4 intermediate results are individually added to the p, r, q, s points to update the values of the p, r, q, s points for the next level. The second stages of both types of *basic* FHT *butterfly* computations involve only 4 additions.

PERMUT	LEVEL = 0	LEVEL = 1	LEVEL = 2	LEVEL = 3	LEVEL = 4	RESULT
H(0) <- H(0)	H(0) <- H(0) + H(1)	H(0) <- H(0) + H(2)C0 + H(2)S0	H(0) <- H(0) + H(4)C0 + H(4)S0	H(0) <- H(0) + H(8)C0 + H(8)S0	H(0) <- H(0) + H(16)C0 + H(16)S0	H(0)
H(1) <- H(16)	H(1) <- H(0) - H(1)	H(1) <- H(1) + H(3)C8 + H(3)S8	H(1) <- H(1) + H(5)C4 + H(7)S4	H(1) <- H(1) + H(9)C2 + H(15)S2	H(1) <- H(1) + H(17)C1 + H(31)S1	H(1)
H(2) <- H(8)	H(2) <- H(2) + H(3)	H(2) <- H(0) + H(2)C16 + H(2)S16	H(2) <- H(2) + H(6)C8 + H(6)S8	H(2) <- H(2) + H(10)C4 + H(14)S4	H(2) <- H(2) + H(18)C2 + H(30)S2	H(2)
H(3) <- H(24)	H(3) <- H(2) - H(3)	H(3) <- H(1) + H(3)C24 + H(3)S24	H(3) <- H(3) + H(7)C12 + H(5)S12	H(3) <- H(3) + H(11)C6 + H(13)S6	H(3) <- H(3) + H(19)C3 + H(29)S3	H(3)
H(4) <- H(4)	H(4) <- H(4) + H(5)	H(4) <- H(4) + H(6)C0 + H(6)S0	H(4) <- H(0) + H(4)C16 + H(4)S16	H(4) <- H(4) + H(12)C8 + H(12)S8	H(4) <- H(4) + H(20)C4 + H(28)S4	H(4)
H(5) <- H(20)	H(5) <- H(4) - H(5)	H(5) <- H(5) + H(7)C8 + H(7)S8	H(5) <- H(1) + H(5)C20 + H(7)S20	H(5) <- H(5) + H(13)C10 + H(11)S10	H(5) <- H(5) + H(21)C5 + H(27)S5	H(5)
H(6) <- H(12)	H(6) <- H(6) + H(7)	H(6) <- H(4) + H(6)C16 + H(6)S16	H(6) <- H(2) + H(6)C24 + H(6)S24	H(6) <- H(6) + H(14)C12 + H(10)S12	H(6) <- H(6) + H(22)C6 + H(26)S6	H(6)
H(7) <- H(28)	H(7) <- H(6) - H(7)	H(7) <- H(5) + H(7)C24 + H(7)S24	H(7) <- H(3) + H(7)C28 + H(5)S28	H(7) <- H(7) + H(15)C14 + H(9)S14	H(7) <- H(7) + H(23)C7 + H(25)S7	H(7)
H(8) <- H(2)	H(8) <- H(8) + H(9)	H(8) <- H(8) + H(10)C0 + H(10)S0	H(8) <- H(8) + H(12)C0 + H(12)S0	H(8) <- H(0) + H(8)C16 + H(8)S16	H(8) <- H(8) + H(24)C8 + H(24)S8	H(8)
H(9) <- H(18)	H(9) <- H(8) - H(9)	H(9) <- H(9) + H(11)C8 + H(11)S8	H(9) <- H(9) + H(13)C4 + H(15)S4	H(9) <- H(1) + H(9)C18 + H(15)S18	H(9) <- H(9) + H(25)C9 + H(23)S9	H(9)
H(10) <- H(10)	H(10) <- H(10) + H(11)	H(10) <- H(8) + H(10)C16 + H(10)S16	H(10) <- H(10) + H(14)C8 + H(14)S8	H(10) <- H(2) + H(10)C20 + H(14)S20	H(10) <- H(10) + H(26)C10 + H(22)S10	H(10)
H(11) <- H(26)	H(11) <- H(10) - H(11)	H(11) <- H(9) + H(11)C24 + H(11)S24	H(11) <- H(11) + H(15)C12 + H(13)S12	H(11) <- H(3) + H(11)C22 + H(13)S22	H(11) <- H(11) + H(27)C11 + H(21)S11	H(11)
H(12) <- H(6)	H(12) <- H(12) + H(13)	H(12) <- H(12) + H(14)C0 + H(14)S0	H(12) <- H(8) + H(12)C16 + H(12)S16	H(12) <- H(4) + H(12)C24 + H(12)S24	H(12) <- H(12) + H(28)C12 + H(20)S12	H(12)
H(13) <- H(22)	H(13) <- H(12) - H(13)	H(13) <- H(13) + H(15)C8 + H(15)S8	H(13) <- H(9) + H(13)C20 + H(15)S20	H(13) <- H(5) + H(13)C26 + H(11)S26	H(13) <- H(13) + H(29)C13 + H(19)S13	H(13)
H(14) <- H(14)	H(14) <- H(14) + H(15)	H(14) <- H(12) + H(14)C16 + H(14)S16	H(14) <- H(10) + H(14)C24 + H(14)S24	H(14) <- H(6) + H(14)C28 + H(10)S28	H(14) <- H(14) + H(30)C14 + H(18)S14	H(14)
H(15) <- H(30)	H(15) <- H(14) - H(15)	H(15) <- H(13) + H(15)C24 + H(15)S24	H(15) <- H(11) + H(15)C28 + H(13)S28	H(15) <- H(7) + H(15)C30 + H(9)S30	H(15) <- H(15) + H(31)C15 + H(17)S15	H(15)
H(16) <- H(1)	H(16) <- H(16) + H(17)	H(16) <- H(16) + H(18)C0 + H(18)S0	H(16) <- H(16) + H(20)C0 + H(20)S0	H(16) <- H(16) + H(24)C0 + H(24)S0	H(16) <- H(0) + H(16)C16 + H(16)S16	H(16)
H(17) <- H(17)	H(17) <- H(16) - H(17)	H(17) <- H(17) + H(19)C8 + H(19)S8	H(17) <- H(17) + H(21)C4 + H(23)S4	H(17) <- H(17) + H(25)C2 + H(31)S2	H(17) <- H(1) + H(17)C17 + H(31)S17	H(17)
H(18) <- H(9)	H(18) <- H(18) + H(19)	H(18) <- H(18) + H(20)C16 + H(18)S16	H(18) <- H(18) + H(22)C8 + H(22)S8	H(18) <- H(18) + H(26)C4 + H(30)S4	H(18) <- H(2) + H(18)C18 + H(30)S18	H(18)
H(19) <- H(25)	H(19) <- H(18) - H(19)	H(19) <- H(17) + H(19)C24 + H(19)S24	H(19) <- H(19) + H(23)C12 + H(21)S12	H(19) <- H(19) + H(27)C6 + H(29)S6	H(19) <- H(3) + H(19)C19 + H(29)S19	H(19)
H(20) <- H(5)	H(20) <- H(20) + H(21)	H(20) <- H(20) + H(22)C0 + H(22)S0	H(20) <- H(16) + H(20)C16 + H(20)S16	H(20) <- H(20) + H(28)C8 + H(28)S8	H(20) <- H(4) + H(20)C20 + H(28)S20	H(20)
H(21) <- H(21)	H(21) <- H(20) - H(21)	H(21) <- H(21) + H(23)C8 + H(23)S8	H(21) <- H(17) + H(21)C20 + H(23)S20	H(21) <- H(21) + H(29)C10 + H(27)S10	H(21) <- H(5) + H(21)C21 + H(27)S21	H(21)
H(22) <- H(13)	H(22) <- H(22) + H(23)	H(22) <- H(20) + H(22)C16 + H(22)S16	H(22) <- H(18) + H(22)C24 + H(22)S24	H(22) <- H(22) + H(30)C12 + H(26)S12	H(22) <- H(6) + H(22)C22 + H(26)S22	H(22)
H(23) <- H(29)	H(23) <- H(22) - H(23)	H(23) <- H(21) + H(23)C24 + H(23)S24	H(23) <- H(19) + H(23)C28 + H(21)S28	H(23) <- H(23) + H(31)C14 + H(25)S14	H(23) <- H(7) + H(23)C23 + H(25)S23	H(23)
H(24) <- H(3)	H(24) <- H(24) + H(25)	H(24) <- H(24) + H(26)C0 + H(26)S0	H(24) <- H(24) + H(28)C0 + H(28)S0	H(24) <- H(16) + H(24)C16 + H(24)S16	H(24) <- H(8) + H(24)C24 + H(24)S24	H(24)
H(25) <- H(19)	H(25) <- H(24) - H(25)	H(25) <- H(25) + H(27)C8 + H(27)S8	H(25) <- H(25) + H(29)C4 + H(31)S4	H(25) <- H(17) + H(25)C18 + H(31)S18	H(25) <- H(9) + H(25)C25 + H(23)S25	H(25)
H(26) <- H(11)	H(26) <- H(26) + H(27)	H(26) <- H(24) + H(26)C16 + H(26)S16	H(26) <- H(26) + H(30)C8 + H(30)S8	H(26) <- H(18) + H(26)C20 + H(30)S20	H(26) <- H(10) + H(26)C26 + H(22)S26	H(26)
H(27) <- H(27)	H(27) <- H(26) - H(27)	H(27) <- H(25) + H(27)C24 + H(27)S24	H(27) <- H(27) + H(31)C12 + H(29)S12	H(27) <- H(19) + H(27)C22 + H(29)S22	H(27) <- H(11) + H(27)C27 + H(21)S27	H(27)
H(28) <- H(7)	H(28) <- H(28) + H(29)	H(28) <- H(28) + H(30)C0 + H(30)S0	H(28) <- H(24) + H(28)C16 + H(28)S16	H(28) <- H(20) + H(28)C24 + H(28)S24	H(28) <- H(12) + H(28)C28 + H(20)S28	H(28)
H(29) <- H(23)	H(29) <- H(28) - H(29)	H(29) <- H(29) + H(31)C8 + H(31)S8	H(29) <- H(25) + H(29)C20 + H(31)S20	H(29) <- H(21) + H(29)C26 + H(27)S26	H(29) <- H(13) + H(29)C29 + H(19)S29	H(29)
H(30) <- H(15)	H(30) <- H(30) + H(31)	H(30) <- H(28) + H(30)C16 + H(30)S16	H(30) <- H(26) + H(30)C24 + H(30)S24	H(30) <- H(22) + H(30)C28 + H(26)S28	H(30) <- H(14) + H(30)C30 + H(18)S30	H(30)
H(31) <- H(31)	H(31) <- H(30) - H(31)	H(31) <- H(29) + H(31)C24 + H(31)S24	H(31) <- H(27) + H(31)C28 + H(29)S28	H(31) <- H(23) + H(31)C30 + H(25)S30	H(31) <- H(15) + H(31)C31 + H(17)S31	H(31)

Figure 3.1. Computational steps in Fast Hartley Transform

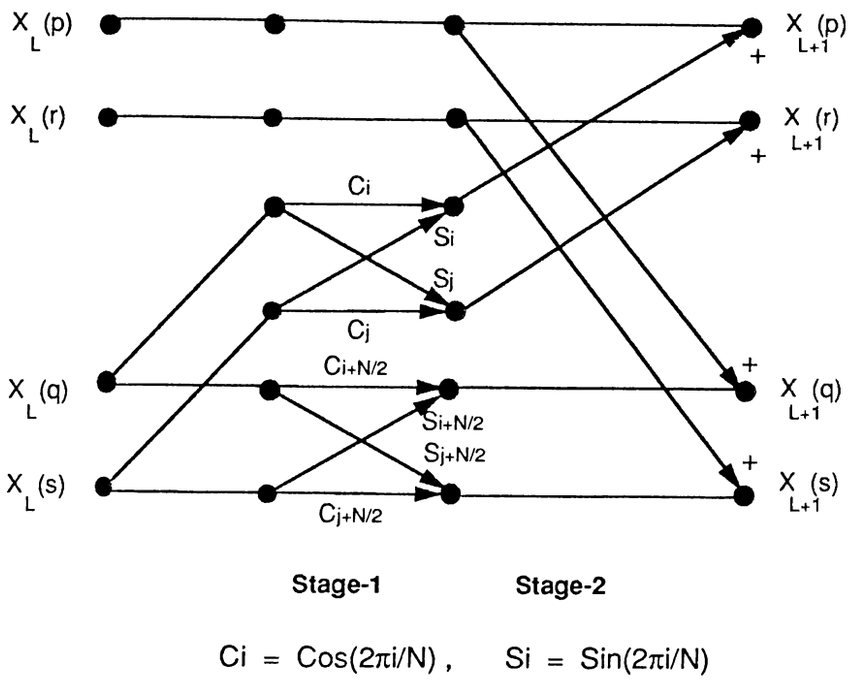

Figure 3.2. Basic Fast Hartley Transform Butterfly, Type1 ($1 \leq \ell \leq$ n-1)

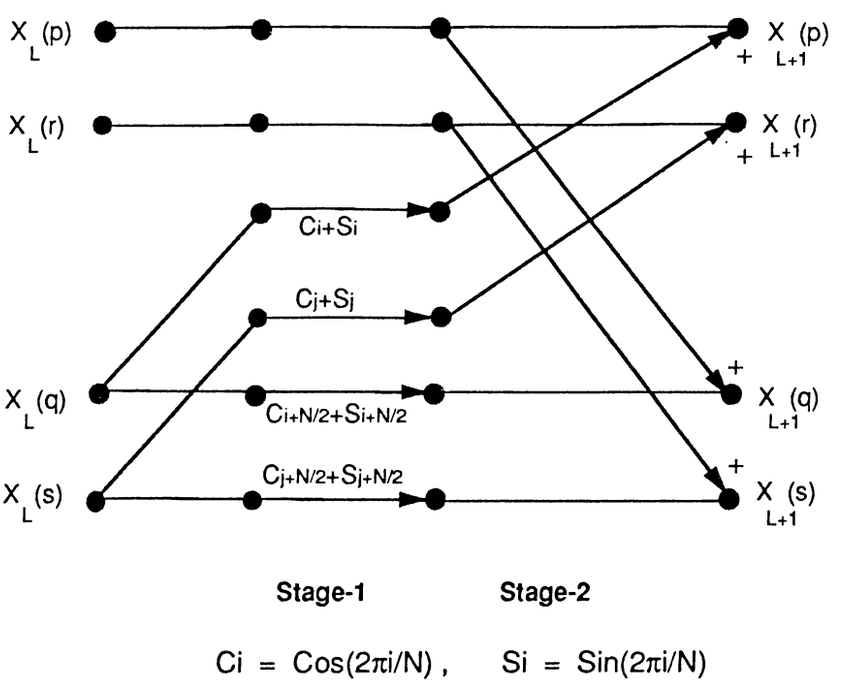

Figure 3.3. Basic Fast Hartley Transform Butterfly, Type2 ($1 \leq \ell \leq$ n-1)

A carefull analysis of *type*-1 *basic FHT butterfly* computation reveals that angles of Cos and Sin factor pairs multiplied by the q-point (i/j and $i+\frac{N}{2}/j+\frac{N}{2}$) and s-point (j/i and $j+\frac{N}{2}/i+\frac{N}{2}$) are mutually π radians away from each other since, $2\pi(i+\frac{N}{2})/N = 2\pi i/N + \pi$ and $2\pi(j+\frac{N}{2})/N = 2\pi j/N + \pi$. Hence,

$$\begin{aligned} X_L(q) \times C_{i+N/2} + X_L(s) \times S_{i+N/2} &= -(X_L(q) \times C_i + X_L(s) \times S_i) \\ X_L(q) \times C_{j+N/2} + X_L(s) \times S_{j+N/2} &= -(X_L(q) \times C_j + X_L(s) \times S_j) \end{aligned} \quad (3.3)$$

since $Cos(x+\pi) = -Cos(x)$ *and* $Sin(x+\pi) = -Sin(x)$. Thus, *type*-1 *basic FHT butterfly* (Fig. 3.2) can be simplified as shown in Fig. 3.4. The resulting FHT butterfly will be referred here as *simplified type*-1 FHT butterfly. The total number of computations in the first stage of a *type*-1 *simplified FHT* butterfly is reduced by a factor of two (from 8 multiplications and 4 additions to 4 multiplications and 2 additions). A similar analysis can also be applied to *type*-2 *basic FHT* butterfly to reduce the number of multiplications involved in the first stage from 4 to 2. Furthermore, a detailed analysis shows that, $Cos + Sin$ factors multiplied by the q and s points are always 1. Hence, the remaining two multiplications can also be omitted. Fig. 3.5 illustrates the computational flow-graph for a *type*-2 *simplified FHT* butterfly. Note that, multiplications with $Cos + Sin$ factors are shown in Fig. 3.5 for the sake of completeness.

The computations involved in a *type*-1 *simplified FHT* butterfly at level-ℓ are as follows:

$$\begin{aligned} qtemp &:= Ci \times X(q) + Si \times X(s) \\ stemp &:= Cj \times X(s) + Sj \times X(q) \\ X(q) &:= X(p) - qtemp \\ X(s) &:= X(r) - stemp \\ X(p) &:= X(p) + qtemp \\ X(r) &:= X(r) + stemp \end{aligned} \quad (3.4)$$

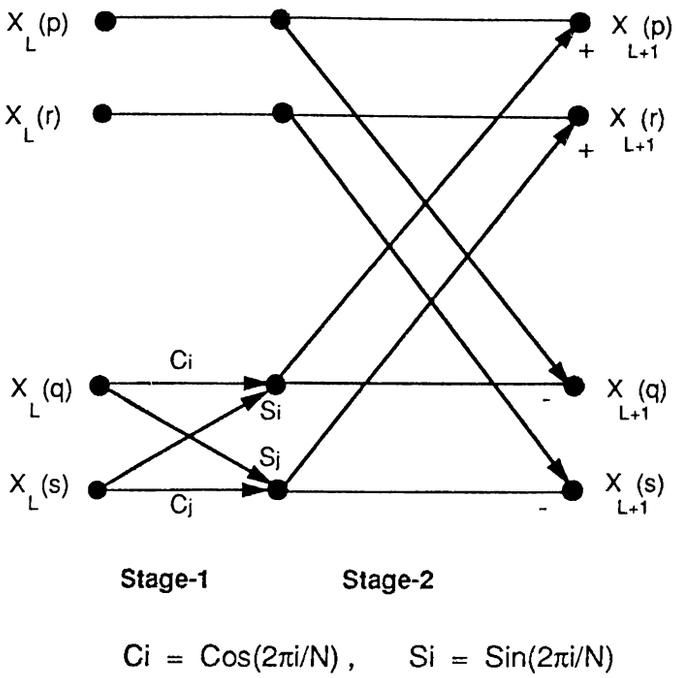

Figure 3.4. Simplified Fast Hartley Transform Butterfly, Type1

The computations involved in a *type-2 simplified FHT* butterfly at level-ℓ are as follows:

$$\begin{aligned} qtemp &:= X(q) \\ stemp &:= X(s) \\ X(q) &:= X(p) - qtemp \\ X(s) &:= X(r) - stemp \\ X(p) &:= X(p) + qtemp \\ X(r) &:= X(r) + stemp \end{aligned} \tag{3.5}$$

For the sake of clarity of the discussions, each FHT point is assumed to have an n-bit binary representation where $n = \log_2 N$. For example, $f_{n-1}, \ldots, f_1, f_0$ denotes the binary representation of an FHT point q, where q denotes its decimal index in the *bit-reversed* ordering. Note that, in both types of *simplified FHT* butterflies, (p, q) and (r, s) pairs differ only in the ℓ^{th} bit of their n-bit binary representation at the ℓ^{th} level such that $q = p + 2^\ell$ and $s = r + 2^\ell$. That is, ℓ^{th} bits of the binary representations of both q and s indices are "1", whereas, ℓ^{th} bits of both p and r indices are "0". Hence, all (p, q) and (r, s) pairs are seperated by 2^ℓ at the ℓ^{th} level.

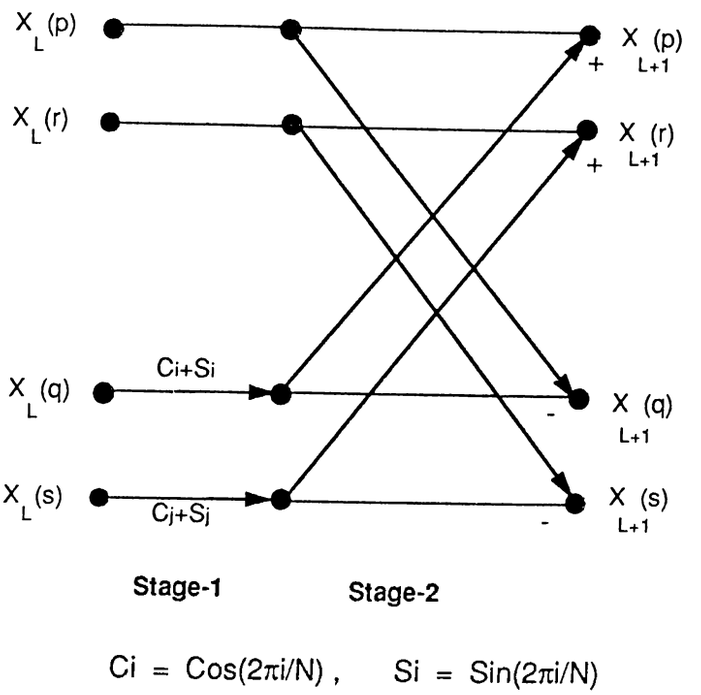

Figure 3.5. Simplified Fast Hartley Transform Butterfly, Type2

In a *type*-1 *simplified FHT* butterfly at the ℓ^{th} level, *FHT* points of each (q, s) pair differ only in the least significant ℓ bits of their n-bit binary representations. This difference is such that, ℓ-1 least significant bits of the binary representations of the q and s indices are mutually 2's complement of each other. That is, $q = f_{n-1}, \ldots, f_{\ell+1}, 1, f_{\ell-1}, \ldots, f_1, f_0$ and $s = f_{n-1}, \ldots, f_{\ell+1}, 1, g_{\ell-1}, \ldots, g_1, g_0$ where "$f_{\ell-1}, \ldots, f_1, f_0$" and "$g_{\ell-1}, \ldots, g_1, g_0$" are mutually ℓ-bit 2's complement of each other. Hence, the seperation between q and s indices of a *type*-1 *simplified FHT* butterfly varies between 2 and 2^{ℓ}-2 at the ℓ^{th} level for $\ell > 2$. In a *type*-2 *simplified FHT* butterfly, at the ℓ^{th} level, q and s points only differ in the $(\ell - 1)^{th}$ bit of their binary representation such that q is a power of 2^{ℓ}, and $s = q + 2^{\ell-1}$. That is, $q = (f_{n-1}, \ldots, f_{\ell+1}, 1, 0, \ldots, 0)$ and $s = (f_{n-1}, \ldots, f_{\ell+1}, 1, 1, 0 \ldots, 0)$. Hence, q and s indices of a *type*-2 *simplified FHT* butterfly are seperated by $2^{\ell-1}$ at the ℓ^{th} level. Hence, *type*-2 *FHT* butterflies at the ℓ^{th} level can easily be identified by 4-tuplets $\{p, q, r, s\} = \{p, p + 2^{\ell-1}, p + 2^{\ell}, p + 3 \times 2^{\ell-1}\}$ where p is a multiple of $2^{\ell+1}$ (i.e., least significant $(\ell + 1)$-bits are all 0's).

The computational flow diagram for the *FHT* algorithm can be derived by exploiting these two types of simplified *FHT* butterflies (Fig. 3.4 and Fig. 3.5).

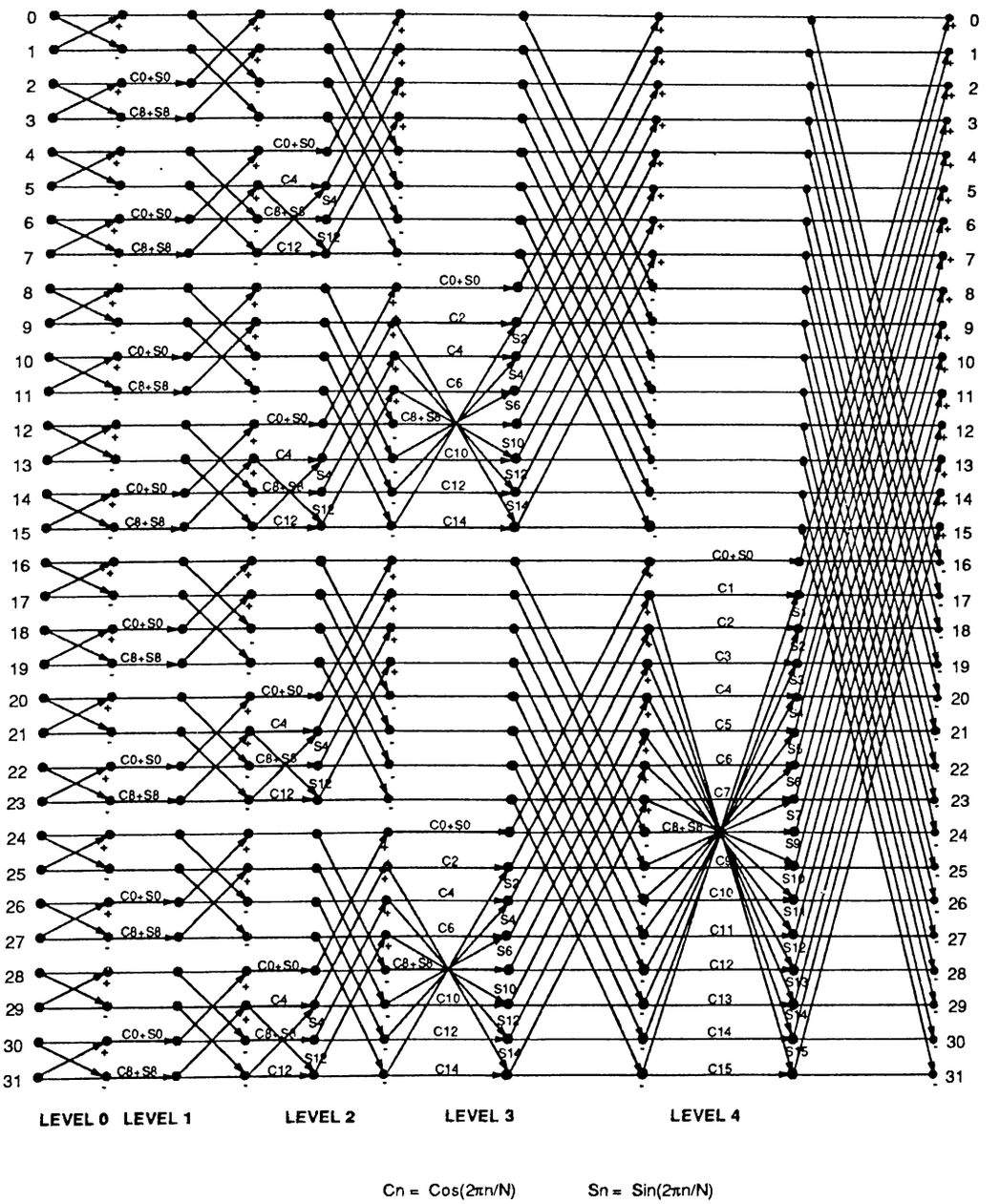

Figure 3.6. Sequential FlowGraph of Fast Hartley Transform for N=32 points

In the rest of the chapter, *simplified FHT* butterflies will be referred as *FHT* butterflies for the sake of simplicity. Fig. 3.6 illustrates the proposed computational flow-graph for (N=32) point *FHT* algorithm. As is seen in Fig. 3.6, first level ($\ell = 0$) is a special level which consists of two-point *FFT* butterflies without any *Cos/Sin* factor multiplications. That is, only addition/subtraction operations are performed in two-point *FFT* butterflies. The following levels ($1 \leq \ell \leq n-1$) consist of $N/2^{\ell+1}$ consecutive groups where each group contains $2^{\ell+1}$ consecutive *FHT* points at the ℓ^{th} level. For example, as is seen in Fig. 3.6, at level ℓ=3, a 32-point *FHT* contains $32/2^{3+1} = 2$ blocks, $G^0_{(3)} = \{0-15\}$, $G^1_{(3)} = \{16-31\}$, where each block consists of $2^{3+1} = 16$ consecutive *FHT* points. The consecutive $2^{\ell+1}$ *FHT* points in each block at level ℓ constitute p, q, r, s points of all $2^{\ell+1}/4 = 2^{\ell-1}$ *FHT* butterflies involved in that group. The first *FHT* point in each block is the p-point of the only *type*-2 *FHT* butterfly in that block. The following $2^{\ell-1}$ *FHT* points in each block constitute the p points of $2^{\ell-1}$ *type*-1 *FHT* butterfly involved in that block. For example, {16,20,24,28} is the only *type*-2 butterfly involved in block $G^1_{(3)} = \{16-31\}$ whereas {17,23,25,31}, {18,22,26,30} and {19,21,27,29} constitute the *type*-1 butterflies in that block. Note that, second level ($\ell = 1$) consists of only $N/4$ *type*-2 *FHT* butterflies and the number of *type*-2 *FHT* butterflies decreases by one half at succeeding levels and reduce to one at the last level ($\ell = n-1$). That is, the number of *type*-2 *FHT* butterflies at the ℓ^{th} level is $N^{\ell}_{t2} = (N/4)/2^{\ell-1} = N/2^{\ell+1}$ for $\ell = 1, 2, \ldots, n-1$. The number of *type*-1 *FHT* butterflies at a particular level (ℓ), $N^{\ell}_{t1} = N(2^{\ell-1}-1)/2^{\ell+1}$ for $\ell = 1, 2, \ldots, n-1$. Note that, $N^{\ell}_{t1} + N^{\ell}_{t2} = N/4$ *FHT* butterflies exist at each level for $1 \leq \ell \leq n-1$.

Prog. 3.1 illustrates the C-like pseudo-code for the computation of N-point *FHT*. In this program, N-real inputs $X(i), (i = 0, 1, \ldots, N-1)$ are assumed to be stored in *bit-reversed* order in one-dimensional X-array of size N. Note that, computations are performed in-place and the results are obtained in the X-array in *normal* order similar to the *FFT* program. The complexity analysis of the sequential *FHT* algorithm is as follows. As is seen in Prog. 3.1, the first *for-loop* in the main program performs the add/sub operations (on two point butterflies) involved in the first level (special level, ℓ=0). Then, *SEQFHT*ℓ

```
/* Input in bit-reversed order in X[0 ... N-1]   */
/* Output in normal order in X[0 ... N-1]       */
n := log2 N
for (i :=0; i < N − 1; i := i + 2) do
    temp := X(i+1)
    X(i + 1) := X(i) − temp
    X(i) := X(i) + temp
endfor
for (ℓ :=1; ℓ < n; ℓ + +) do
    Call SEQFHTℓ(X,CSfac,N,ℓ)
endfor

SEQFHTℓ(X,CSfac,N,ℓ)
    for (i := 0; i < N/2^(ℓ+1)-1; i++) do
        p := i × 2^(ℓ+1);  q := p + 2^ℓ;  r := p + 2^(ℓ−1);  s := q + 2^(ℓ−1);
        qtemp := X(q)
        stemp := X(s)
        X(q) := X(p) − qtemp
        X(s) := X(r) − stemp
        X(p) := X(p) + qtemp
        X(r) := X(r) + stemp
        for (j := 1; j < 2^(ℓ−1) − 1; j++) do
            p := i × 2^(ℓ+1) + j;  q := p + 2^ℓ;
            r := (i × 2^(ℓ+1) + 2^ℓ) − j;  s := r + 2^ℓ;
            qtemp := Cfac1 × X(q) + Sfac1 × X(s)
            stemp := Cfac2 × X(s) + Sfac2 × X(q)
            X(q) := X(p) − qtemp
            X(s) := X(r) − stemp
            X(p) := X(p) + qtemp
            X(r) := X(r) + stemp
        endfor
    endfor
endSEQFHTℓ
```

Program 3.1 : Sequential N-pt FHT Algorithm

function is invoked $n-1$ times to perform the FHT computations at the succeding levels ($1 \leq \ell \leq n-1$). The outer *for-loop* in $SEQFHT\ell$ function iterates $N/2^{\ell+1}$ times to identify the $N/2^{\ell+1}$ consecutive FHT blocks. The p, q, r, s indices of the only *type*-2 FHT butterfly in a particular block are identified at the first line inside the outer *for-loop*. The computations involved in that *type*-2 FHT butterfly are performed in the following six statements. The inner *for-loop* iterates $2^{\ell-1}$ times to identify and perform the computations involved in $2^{\ell-1}$ *type*-1 FHT butterflies in that group. The total number of *type*-1 and *type*-2 FHT butterfly computations are

$$\sum_{\ell=1}^{n-1} N_{t1}^{\ell} = \sum_{\ell=1}^{n-1} N(2^{\ell-1}-1)/2^{\ell+1} = \frac{N}{4}(\log_2 N - 1) - \frac{N}{2} + 1 \qquad (3.6)$$

$$\sum_{\ell=1}^{n-1} N_{t2}^{\ell} = \sum_{\ell=1}^{n-1} N/2^{\ell+1} = \frac{N}{2} - 1 \qquad (3.7)$$

succesively. Recalling that *type*-1 and *type*-2 FHT butterfly computations require 10 and 4 floating-point operations succesively, the overall time complexity of an N-point FHT computation is

$$T_{P1} = [2.5N \log_2 N - 4.5N + 6]\, t_{calc} \qquad (3.8)$$

where t_{calc} is the time required for a real floating-point multiplication or addition. The computation of Cos/Sin factors are not involved in the above complexity analysis as in the case of FFT. It is assumed that, $N/2$ Cos/Sin coefficients are calculated before and stored in a table ($CSfac$ in $SEQFHT\ell$ function) prior to the execution of an N-point FHT problem instance.

3.3 Parallel FHT Algorithm

The static mapping scheme presented (for parallel FFT) in Chapter 2 can also be applied for the distribution of FHT data and computations. Recall that, in this mapping scheme, succesive processors in the decimal ordering are assigned the succesive slices of the X-array with each slice containing $M = N/P = 2^n/2^d = 2^{n-d}$ consecutive FHT points. The decomposition of the 32-point FHT data and computations on a 3-dimensional hypercube, with $M = 4$ FHT points assigned to each processor is illustrated in Fig. 3.7.

In this scheme, each processor is responsible for carrying out the complete in-place computations required for the local M FHT points assigned to itself. For the sake of clarity of the discussions in the following sections, each processor P_i is assumed to have a d-bit binary representation $b_{d-1}, b_{d-2}, \ldots, b_1, b_0$ in a d-dimensional hypercube with $P = 2^d$ processors. Here, P_i denotes the decimal index of a particular processor.

Recall that, FHT butterflies are confined within consecutive blocks of lengths $2^{\ell+1}$ at level-ℓ. Hence, computational interdependencies are confined within consecutive groups of lengths $2, 4, 8, \ldots, 2^{n-d}$ during the first $n-d$ levels $\ell = 0, 1, \ldots, n-d-1$. Thus, no interprocessor communication is required during the first $n-d$ levels since consecutive $M = N/P = 2^{n-d}$ FHT points are assigned in groups to consecutive processors. However, in the last d-levels, FHT butterflies will be fragmented between processors thus neccesitating interprocessor communication. As is seen in Fig. 3.4 and Fig. 3.5 there exists computational interdependencies between p and q points and between r and s points during the second stages of both types of FHT butterflies. In the last d-levels, both (p, q) and (r, s) pairs are seperated by $2^{n-d}, 2^{n-d-1}, \ldots, 2^{n-1}$ respectively such that p and q points of the (p, q) pairs and the r and s points of the (r, s) pairs are assigned to neighbor processor pairs whose indices differ by $2^0, 2^1, \ldots, 2^{d-1}$ respectively. Hence, one concurrent pairwise exchange communication step is required just before the second stage computations of each level of the last d levels in order to exchange p and r values with q and s values respectively, and vice-versa. Note that, all processors should involve in these pairwise exchange communication operations at each 2^{nd} stage of the last d levels. The volume of information exchange between each processor pair is M FHT points.

As is seen in Fig. 3.4, there exists computational interdependencies between the q and s points during the first stage computations of *type*-1 FHT butterflies. As is seen in Fig. 3.7, these interdependencies are unsymmetrical in nature and hence the nature of interprocessor communication necessitated

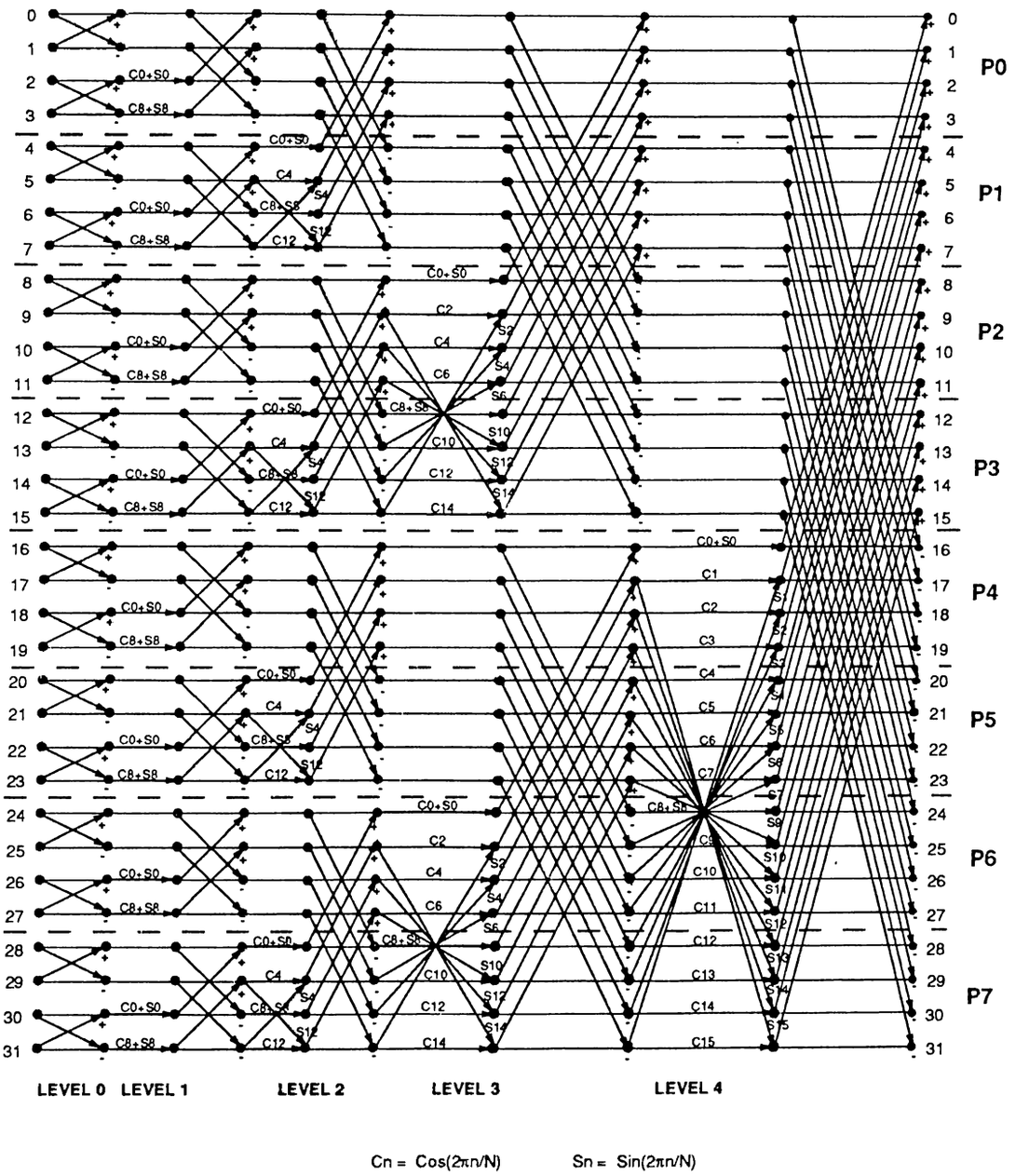

Figure 3.7. Static mapping of an (N=32) point Fast Hartley Transform on a 3-dimensional hypercube

by these interdependencies are more complicated. The q and s points are confined to the second half of each FHT group. Note that, at level $\ell = n - d$, odd-numbered processors ($b_0 = 0$) will be assigned the second halves of the consecutive FHT blocks. Hence, odd-numbered processors will hold all (q, s) pairs at level $\ell = n - d$ thus avoiding the interprocessor communication. However, in the last $d-1$ levels, ($\ell = n-d+1, \ldots, n-1$) q and s points of the (q, s) pairs will also be fragmented between processor pairs. In these levels, only one half of the processors ($b_{\ell-(n-d)} = 1$) hold q and s points. The assignment is such that half of the processors ($b_{\ell-(n-d)} = 1$, $b_{\ell-(n-d)-1} = 0$) hold either M or $M - 1$ q-points of *type*-1 FHT butterflies and the other half hold ($b_{\ell-(n-d)} = 1$, $b_{\ell-(n-d)-1} = 1$) hold either M or $M - 1$ s-points of *type*-1 FHT butterflies. Those processors ($b_{\ell-(n-d)}, b_{\ell-(n-d-1)}, \ldots, b_0$="100...0") holding $M-1$ q points of *type*-1 FHT butterflies hold a single q point of a *type*-2 FHT butterfly as the first data point of their local M points. Similarly, those processors ($b_{\ell-(n-d)}, b_{\ell-(n-d)-1}, \ldots, b_0$="110...0") holding $M - 1$ s-points of *type*-1 FHT butterflies hold a single s point of a *type*-2 FHT butterfly as the first data point of their local M FHT points. The fragmentation of q and s points of *type*-1 FHT butterflies is such that q and s points in the last $M - 1$ local FHT data are assigned in groups to processor pairs (P_i, Pj) whose indices differ only in the least significant $(\ell - (n - d))$ bits of their d-bit binary representation. That is, if $P_i = (b_{d-1}, \ldots, b_{\ell-(n-d)+1}, 1, b_{\ell-(n-d)-1}, \ldots, b_1, b_0)$ then $P_j = (b_{d-1}, \ldots, b_{\ell-(n-d)+1}, 1, b'_{\ell-(n-d)-1}, \ldots, b'_1, b'_0)$ where b'_i denotes the complement of bit b_i.

At level $\ell = (n - d) + 1$, this fragmentation requires $P/4$ exchange communication operations between neighbor processor pairs whose indices differ only in the least significant bit, b_0. However, in the succeeding levels $\ell = (n-d)+2, (n-d)+3, \ldots, n-1$, $P/4$ exchange communication operations will take place between non-neighbor processors with multi-hop communication distances $2, 3, \ldots, d-1$ respectively. The volume of information exchange during each pairwise communication operation is $M - 1$ real FHT points for those processors to gather $M - 1$ (q, s) pairs to perform their part of the first stage computations of *type*-1 FHT butterflies at that level.

At level ℓ, $2^{n-\ell}$ and $P/2-2^{n-\ell}$ processors will be assigned $M-1$ and M q or s points of *type*-1 FHT butterflies respectively. Hence, at level $\ell = (n-d)+1$, there will be no processors holding M s or q points of *type*-1 FHT butterflies. Thus, pairwise exchange communication step mentioned above will be sufficient for those processors to perform their part of *type*-1 FHT computations involved in the first stage of that level. However, at succeeding levels, $\ell = (n-d)+2, (n-d)+3, \ldots, n-1$ there will be $2^{d-2}, 3\times 2^{d-3}, \ldots, P/2-2$ processors holding M q or s points of *type*-1 FHT butterflies. Hence, each one of these processors need to perform one more exchange communication to gather the last necessary (q,s) pairs of *type*-1 FHT butterfly.

The nature of these exchange communication operations is as follows: if processor pair (P_i, P_j) exchanges their last $(M-1)$ FHT points, then processor pair (P_{i+1}, P_j) will exchange their first local FHT data point. Those processor pairs, (P_i, P_j) for which $P_j = P_{i+1}$ do not need to communicate. In fact, those processor pairs hold only $(M-1)$ (q,s) points of *type*-1 FHT butterflies. Some of these communications will be single-hop, (nearest-neighbor), and some others will be multi-hop, (between non-neighbor processors), Note that, the volume of each pairwise communication operation will be a single FHT point.

It is clear that, $2d-3$ concurrent exchange communication steps are required for the parallel computations involved in the first stages of the last $d-1$ levels. As is mentioned earlier, most of these pairwise exchange communication operations are between non-neighbor processor pairs. Recall that, d concurrent exchange communication (between neighbor processor pairs) are also required for the second stage of last $d-1$ levels in order to complete the FHT butterfly computations. The total number of concurrent communication operations are therefore $3d-3$.

Note that, during the first $n-d$ levels, $(\ell = 0, 1, \ldots, n-d-1)$ each processor will be assigned equal number $(2^{n-d-\ell-1})$ of FHT groups. Thus, each

processor will be assigned exactly equal number of FHT butterflies since each group contains equal number of FHT butterflies. Hence, the static mapping scheme achieves perfect load balance during the first $(n-d)$ levels. However, load balance is disturbed during the parallel first-stage computations involved in the last d-levels. Note that, only q and s points of *type*-1 FHT butterflies are involved in first stage computations. As is mentioned earlier, only half of the processors ($b_{\ell-(n-d)} = 1$) hold $(M-1)$ or M, q or s points whereas the other half ($b_{\ell-(n-d)} = 0$) hold $(M-1)$ or M, p or r points of *type*-1 FHT butterflies. Hence, those processors holding q or s points perform $6(M-1)$ or $6M$ floating-point operations and the other processors wait idle for receiving first stage computation results from those processors.

3.3.1 Perfect Load Balance

A mapping scheme which achieves perfect-load balance is proposed by Lin [18] by introducing the Hartley graph concept. Lin's algorithm is a dynamic mapping scheme which effectively exchanges the responsibility of the further FHT computations associated with half of the exchanged FHT points. The other halves of the FHT points exchanged between processor pairs is due to the fragmentation of FHT butterflies. In Lin's mapping algorithm, the (p,r) and (q,s) pairs of each level-ℓ, *type*-1 FHT butterfly are updated at level $(\ell-1)$ by a processor pair P_i and P_j respectively, which are neighbors on the Hartley graph. Then, at level-ℓ, processors P_i and P_j becomes responsible for computing the (p,s) and (r,q) pairs of level-ℓ, *type*-1 FHT butterflies respectively. Hence, Lin's mapping algorithm necessitates only d-concurrent exchange communication steps (during the first d-levels) since it prevents the fragmentation of the (q,s) pairs of *type*-1 FHT butterflies. However, Lin's algorithm achieves perfect-load balance only for the FHT algorithm which uses basic FHT butterflies. Load balance is disturbed if *simplified* FHT butterflies are exploited due to the unsymmetrical fragmentation of FHT butterflies between processor pairs. Furthermore, the volume of each exchange communication operation is M FHT points due to the unsymmetrical fragmentation of FHT butterflies. As is also indicated in [18], exchange communications are not always between

neighbor processors of the hypercube since Hartley graph cannot be embedded with dilation-one onto the hypercube graph.

In this section, we propose a new computational structure for the sequential FHT algorithm using simplified FHT butterflies. This re-structuring involves re-ordering of FHT points between succesive levels. Then, by exploiting this re-structuring, we will propose a dynamic mapping scheme which achieves the following; (i) perfect load balance for simplified FHT butterfly computations, (ii) only d concurrent exchange communication steps between nearest neighbor processor pairs, (iii) only $M/2 = N/2P$ FHT point exchange in each exchange communication operation.

Restructuring

The computational interdependencies between the successive levels of the FHT algorithm should be closely examined in order to achieve a suitable restructuring for an efficient parallelization. Note that, two consecutive blocks $G^{2i}_{(\ell)}$ and $G^{2i+1}_{(\ell)}$ $(i = 0, 1, \ldots, 2^\ell)$ at level-ℓ constitute the block $G^{i}_{(\ell+1)}$ at the next level-$(\ell+1)$. Also note that, the $(\ell+1)^{th}$ bits of all $2^{\ell+1}$ FHT points in block $G^{2i}_{(\ell)}$ are 0's whereas $(\ell+1)^{th}$ bits of all $2^{\ell+1}$ FHT points in the consecutive block $G^{2i+1}_{(\ell)}$ are 1's. We can deduce the following two facts by considering the butterfly pairs $(B^0_{(\ell)} \in G^{2i}_{(\ell)}, B^1_{(\ell)} \in G^{2i+1}_{(\ell)})$ where $B^1_{(\ell)} - B^0_{(\ell)} = 2^{\ell+1}$. Here, $B^1_{(\ell)} - B^0_{(\ell)} = 2^{\ell+1}$ denotes that, $p^1_\ell - p^0_\ell = r^1_\ell - r^0_\ell = q^1_\ell - q^0_\ell = s^1_\ell - s^0_\ell = 2^{\ell+1}$ where $B^1_{(\ell)} = \{p^1_\ell, r^1_\ell, q^1_\ell, s^1_\ell\}$ and $B^0_{(\ell)} = \{p^0_\ell, r^0_\ell, q^0_\ell, s^0_\ell\}$.

Fact1: Each level-ℓ, *type*-1 FHT butterfly pair $(B^0_{t1(\ell)} \in G^{2i}_{(\ell)}, B^1_{t1(\ell)} \in G^{2i+1}_{(\ell)})$, where $B^1_{t1(\ell)} - B^0_{t1(\ell)} = 2^{\ell+1}$, constitute the *type*-1 butterfly pair $(B1_{t1(\ell)}, B2_{t1(\ell)}) \in G^2_{(\ell+1)}$ at the next level-$(\ell+1)$, where $B1_{t1(\ell+1)} = \{p1_{(\ell+1)}, r1_{(\ell+1)}, q1_{(\ell+1)}, s1_{(\ell+1)}\} = \{p^0_\ell, s^0_\ell, p^1_\ell, s^1_\ell\}$ and $B2_{t1(\ell+1)} = \{p2_{(\ell+1)}, r2_{(\ell+1)}, q2_{(\ell+1)}, s2_{(\ell+1)}\} = \{r^0_\ell, q^0_\ell, r^1_\ell, q^1_\ell\}$. See Fig. 3.8.

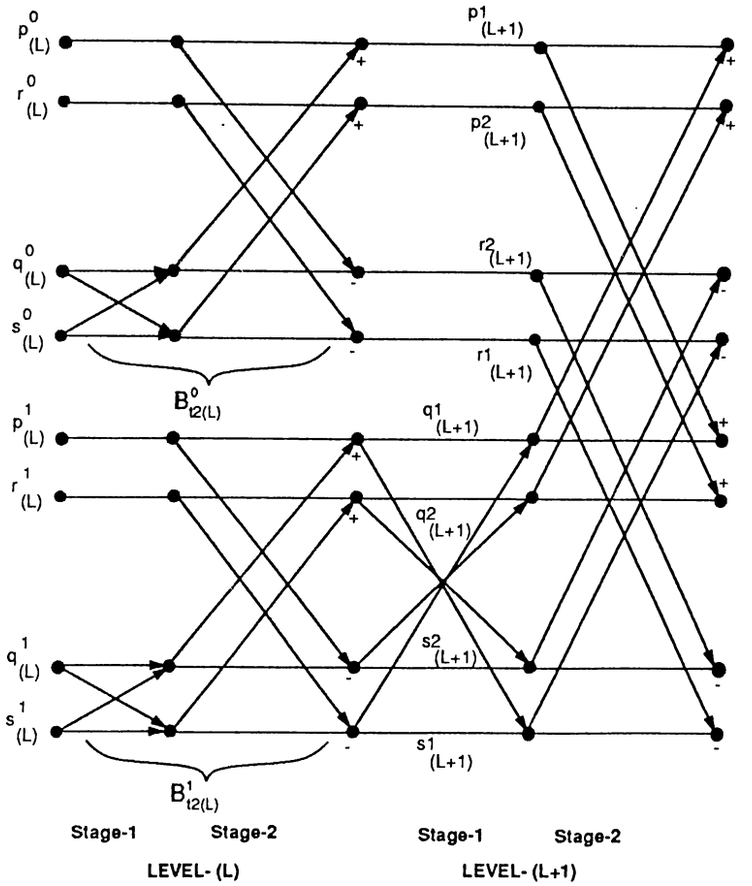

Figure 3.8. Two steps of Simplified Fast Hartley Transform Butterfly, Type1

Proof: The FHT points in (p_ℓ^0, p_ℓ^1) and (s_ℓ^0, s_ℓ^1) pairs of $B1_{t1(\ell+1)} = \{p_\ell^0, s_\ell^0, p_\ell^1, s_\ell^1\}$ differ only in their $(\ell+1)^{th}$ bits such that $p_\ell^1 = p_\ell^0 + 2^{\ell+1}$, $s_\ell^1 = s_\ell^0 + 2^{\ell+1}$ since $B_{t1(\ell)}^1 - B_{t1(\ell)}^0 = 2^{\ell+1}$. Least significant ℓ bits of p_ℓ^1 and s_ℓ^1 are mutually 2's complement of each other and ℓ^{th} bits of p_ℓ^1 and s_ℓ^1 are complement of each other since $B_{t1(\ell)}^1$ is a *type*-1 FHT butterfly at level-ℓ. Hence, least significant $(\ell+1)$ bits of p_ℓ^1 and s_ℓ^1 are mutually 2's complement of each other. Thus, the 4-tuplets $\{p_\ell^0, s_\ell^0, p_\ell^1, s_\ell^1\}$ constitutes a *type*-1 FHT butterfly at level-$(\ell+1)$. Similar steps can be followed to prove that the 4-tuplets $\{r_\ell^0, q_\ell^0, r_\ell^1, q_\ell^1\}$ also constitutes a *type*-1 FHT butterfly at level-$(\ell+1)$.

Fact2: Each level-ℓ, *type*-2 FHT butterfly pair $(B_{t2(\ell)}^0 \in G_{t2(\ell)}^{2i}, B_{t2(\ell)}^1 \in G_{t2(\ell)}^{2i+1})$ constitute the butterfly pair $(B1_{t2(\ell+1)}, B2_{t2(\ell+1)}) \in G_{\ell+1}^i$ at the next level-$(\ell+1)$, where $B1_{t2(\ell+1)} = \{p1_{(\ell+1)}, r1_{(\ell+1)}, q1_{(\ell+1)}, s1_{(\ell+1)}\} = \{p_\ell^0, q_\ell^0, p_\ell^1, q_\ell^1\}$ is a *type*-2 butterfly and

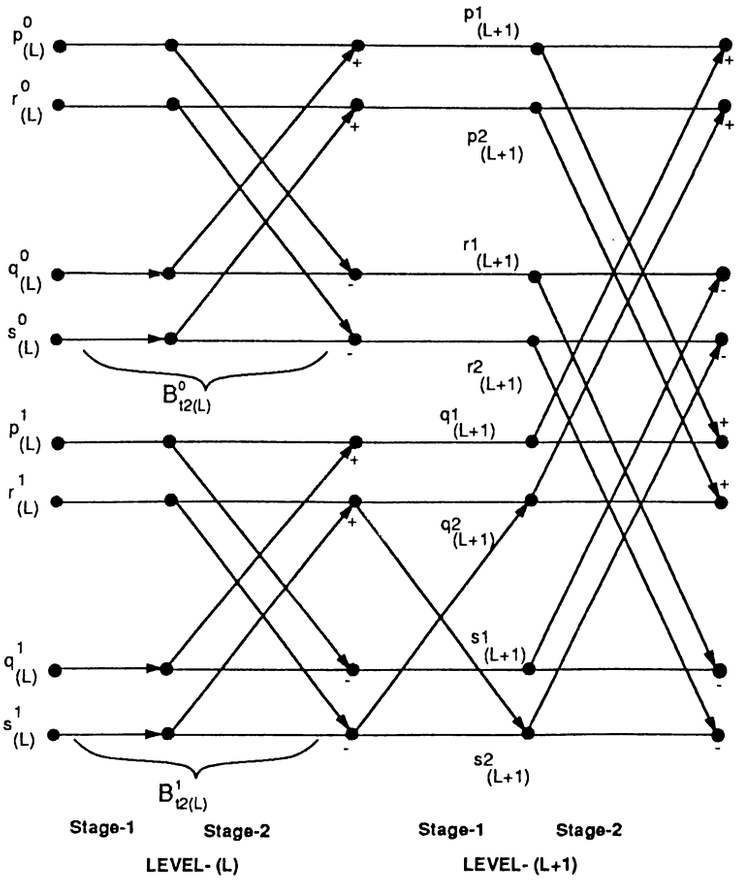

Figure 3.9. Two steps of Simplified Fast Hartley Transform Butterfly, Type2

$B2_{t1(\ell+1)} = \{p2_{(\ell+1)}, r2_{(\ell+1)}, q2_{(\ell+1)}, s2_{(\ell+1)}\} = \{r^0_\ell, s^0_\ell, r^1_\ell, s^1_\ell\}$ is a *type*-1 butterfly. See Fig. 3.9.

Proof: (i) The (p^0_ℓ, p^1_ℓ) and (s^0_ℓ, s^1_ℓ) pairs of $B1_{t2(\ell+1)} = \{p^0_\ell, q^0_\ell, p^1_\ell, q^1_\ell\}$ satisfy $p^1_\ell = p^0_\ell + 2^{\ell+1}$ and $q^1_\ell = q^0_\ell + 2^{\ell+1}$ since $B^1_{t2(\ell)} - B^0_{t2(\ell)} = 2^{\ell+1}$. However, $q^0_\ell = p^1_\ell + 2^\ell$ and $q^1_\ell = p^1_\ell + 2^\ell$ since $B^0_{t2(\ell)}$ and $B^1_{t2(\ell)}$ are level-ℓ, *type*-2 FHT butterflies. Hence, the 4-tuplets $\{p^0_\ell, q^0_\ell, p^1_\ell, q^1_\ell\} = B1_{t1(\ell+1)}$, is a *type*-2 FHT butterfly at level-$(\ell+1)$ since it satisfies $q^0_\ell = p^0_\ell + 2^\ell$, $p^1_\ell = p^0_\ell + 2^{\ell+1}$, $q^1_\ell = p^0_\ell + 2^{\ell+1}$ $= p^0_\ell + 3 \times 2^\ell$. (ii) The pair of points in (r^0_ℓ, r^1_ℓ) and (s^0_ℓ, s^1_ℓ) pairs of $B2_{t1(\ell+1)}$ $= \{r^0_\ell, s^0_\ell, r^1_\ell, s^1_\ell\}$ differ only in their $(\ell+1)^{th}$ bit such that $r^1_\ell = r^0_\ell + 2^{\ell+1}$ and $s^1_\ell = s^0_\ell + 2^{\ell+1}$ since $B^1_{t2(\ell)} - B^0_{t2(\ell)} = 2^{\ell+1}$. The least significant $(\ell+1)^{th}$ bits of r^1_ℓ and s^1_ℓ are "010...0" and "110...0" since $B^1_{t2(\ell)}$ is a *type*-2 FHT butterfly. Hence, least significant $(\ell+1)$ bits of r^1_ℓ and s^1_ℓ are mutually 2's complement of

each other. Thus, the 4-tuplets $\{r_\ell^0, s_\ell^0, r_\ell^1, s_\ell^1\} = B2_{t1(\ell+1)}$, constitute a *type*-1 FHT butterfly at level-$(\ell+1)$.

In the discussions given so far, p, q, r and s labels were used both to identify different points of FHT butterflies and the decimal indices of the corresponding FHT points in the X-array. However, for the sake of clarity of further discussions, p, q, r and s labels will be used only to identify different points of FHT butterflies, whereas i and j labels will be used to identify their decimal indices in the X-array. For example, we will consider the combination structures of the butterfly pairs $(B_{(\ell)}^0 \in G_{(\ell)}^{2i}, B_{(\ell)}^1 \in G_{(\ell)}^{2i+1})$ where $B_{(\ell)}^0 = \{p_\ell^0, r_\ell^0, q_\ell^0, s_\ell^0\} = \{i_1, i_2, i_3, i_4\}$ and $B_{(\ell)}^1 = \{p_\ell^1, r_\ell^1, q_\ell^1, s_\ell^1\} = \{j_1, j_2, j_3, j_4\}$. Note that, i and j parameters satisfy the same relations previously defined for p, r, q and s points. (i.e., $i_3 = i_1 + 2^\ell, i_4 = i_2 + 2^\ell, \ldots$, etc) In this notation, Fact 1 reveals that, $(B1_{t1(\ell+1)}, B2_{t2(\ell+1)})$ pairs generated by *type*-1 $(B_{t1(\ell)}^0, B_{t1(\ell)}^1)$ pairs will have the following structure in the X-array; $B1_{t1(\ell+1)} = \{i_1, i_4, j_1, j_4\}$ and $B2_{t1(\ell+1)} = \{i_2, i_3, j_2, j_3\}$. Similarly, Fact 2 reveals that, $(B1_{t2(\ell+1)}, B2_{t1(\ell+1)})$ pairs generated by *type*-2 $(B_{t2(\ell)}^0, B_{t2(\ell)}^1)$ pairs will have the following structure in the X-array; $B1_{t2(\ell+1)} = \{i_1, i_3, j_1, j_3\}$ and $B2_{t1(\ell+1)} = \{i_2, i_4, j_2, j_4\}$. For example, in a 32-point FHT algorithm, the *type*-1 butterfly pair ($\{1, 7, 9, 15\} \in G_{(3)}^2$, $\{17, 23, 25, 31\} \in G_{(3)}^3$) at level ℓ=3, constitute the *type*-1 butterfly pair ($\{1, 15, 17, 31\}$, $\{7, 9, 23, 25\}$) $\in G_{(4)}^1$) at the next level $\ell = 4$. Similarly, the *type*-2 butterfly pair ($\{0, 4, 8, 12\} \in G_{(3)}^2$, $\{16, 20, 24, 28\} \in G_{(3)}^3$) at level ℓ=3, constitute the (*type*-2, *type*-1) butterfly pair ($\{0, 8, 16, 24\}$, $\{4, 12, 20, 28\}$) $\in G_{(4)}^1$ at the next level $\ell = 4$. Note that, FHT points of both *type*-1 and *type*-2 FHT butterfly pairs at a particular level are scrambled to constitute the FHT points of butterfly pairs at the next level. Also note that, the combination structures of *type*-1 and *type*-2 butterfly pairs are different compared to each other.

As is mentioned earlier, $2d - 3$ concurrent exchange communication operations required during the last $d - 1$ levels of the static mapping scheme are due to the fragmentation of q and s points of *type*-1 FHT butterflies between (usually non-neighbor) processor pairs. This fragmentation occurs due

to the irregular seperation between q and s points of FHT butterflies. Fact1 and Fact2 reveal that regularly seperated (by powers of 2's) butterfly pairs at a particular level constitute scrambled FHT butterfly pairs at the following level. The scrambled combination of the FHT butterfly pairs is the main reason for the irregular spacing between q and s points of FHT butterflies in the following levels. However, this scrambling between FHT butterfly pairs can be avoided by a clever re-ordering while storing the computational results of each FHT butterfly into the X-array. This internal re-ordering will be different for *type*-1 and *type*-2 FHT butterfly pairs since the combination structures of these two types of butterfly pairs are different from each other. Combination structure of *type*-2 FHT butterfly pairs are also investigated since they generate a single *type*-1 FHT butterfly at the folowing level.

The scrambled combination of *type*-1 butterfly pairs can be avoided by swapping r and s points of *type*-1 butterflies while storing the updated values of these FHT points into the X-array. In this scheme, the results of *type*-1 $(B^0_{t1(\ell)}, B^1_{t1(\ell)})$ pairs will have the following order in the X-array at the completion of level-ℓ computations; $B1^0_{t1(\ell)} = \{i_1, i_4, i_3, i_2\}$ and $B1^1_{t1(\ell)} = \{j_1, j_4, j_3, j_2\}$. Recall that, first and last FHT points of $B^0_{t1(\ell)}$ and $B^1_{t1(\ell)}$ butterflies constitute $B1_{t1(\ell+1)}$ and middle two FHT points of $B^0_{t1(\ell)}$ and $B^1_{t1(\ell)}$ butterflies constitute $B2_{t1(\ell+1)}$ at the next level. Hence, in the proposed scheme, *type*-1 $(B1_{t1(\ell+1)}, B2_{t1(\ell+1)})$ pairs will have the following structure in the X-array. $B1_{t1(\ell+1)} = \{p1_{(\ell+1)}, r1_{(\ell+1)}, q1_{(\ell+1)}, s1_{(\ell+1)}\} = \{i_1, i_2, j_1, j_2\}$ and $B2_{t1(\ell+1)} = \{p2_{(\ell+1)}, r2_{(\ell+1)}, q2_{(\ell+1)}, s2_{(\ell+1)}\} = \{i_4, i_3, j_4, j_3\}$. Fig. 3.10 illustrates the proposed restructuring operation during the computation of *type*-1 FHT butterflies.

The scrambled combination of *type*-2 butterfly pairs can be similarly avoided by swapping r and q points of *type*-2 butterflies while storing the updated values into the X-array. In this scheme, the results of *type*-2 $(B^0_{t2(\ell)}, B^1_{t2(\ell)})$ pairs will have the following order in the X-array at the completion of level-ℓ computations; $B^0_{t2(\ell)} = \{i_1, i_3, i_2, i_4\}$ and $B^1_{t2(\ell)} = \{j_1, j_3, j_2, j_4\}$. Recall that, first and third FHT points of $B^0_{t2(\ell)}$ and $B^1_{t2(\ell)}$ butterflies constitute $B1_{t2(\ell+1)}$ and second

and fourth FHT points of $B^0_{t2(\ell)}$ and $B^1_{t2(\ell)}$ butterflies constitute $B2_{t1(\ell+1)}$ at the next level. Hence, in the proposed scheme, $(B1_{t2(\ell+1)}, B2_{t1(\ell+1)})$ pairs will have the following structure in the X-array; $B1_{t2(\ell+1)} = \{p1_{(\ell+1)}, r1_{(\ell+1)}, q1_{(\ell+1)}, s1_{(\ell+1)}\}$ $= \{i_1, i_2, j_1, j_2\}$ and $B2_{t1(\ell+1)} = \{p2_{(\ell+1)}, r2_{(\ell+1)}, q2_{(\ell+1)}, s2_{(\ell+1)}\} = \{i_4, i_3, j_4, j_3\}$. Fig. 3.11 illustrates the proposed re-structuring operation during the computation of *type*-2 FHT butterflies.

The computations involved in a restructured *type*-1 *simplified* FHT butterfly at level-ℓ are as follows:

$$\begin{aligned} qtemp &:= Ci \times X(q) + Si \times X(s) \\ stemp &:= Cj \times X(s) + Sj \times X(q) \\ X(q) &:= X(p) - qtemp \\ X(p) &:= X(p) + qtemp \\ X(s) &:= X(r) + stemp \\ X(r) &:= X(r) - stemp \end{aligned} \tag{3.9}$$

The computations involved in a restructured *type*-2 *simplified* FHT butterfly at level-ℓ are as follows:

$$\begin{aligned} qtemp &:= X(q) \\ stemp &:= X(s) \\ X(s) &:= X(r) - stemp \\ X(q) &:= X(r) + stemp \\ X(r) &:= X(p) - qtemp \\ X(p) &:= X(p) + qtemp \end{aligned} \tag{3.10}$$

The proposed re-structuring has the following nice features. The combination structures of both types of butterfly pairs are very similar. Assume that, level-ℓ butterfly pair $(B^0_{(\ell)}, B^1_{(\ell)})$ constitute the butterfly pair $(B1^0_{(\ell+1)}, B2^1_{(\ell+1)})$ at the next level-$(\ell+1)$. The first/last two FHT points of B^0_ℓ followed by the first/last two FHT points of B^1_ℓ will constitute $B1_{(\ell+1)}/B2_{(\ell+1)}$ respectively, at the next level. Note that, the proposed re-structuring avoids the scrambled combination structure between butterfly pairs at succesive levels. Furthermore, in the proposed scheme, (p, r) points and (q, s) points of both $B1_{(\ell+1)}$

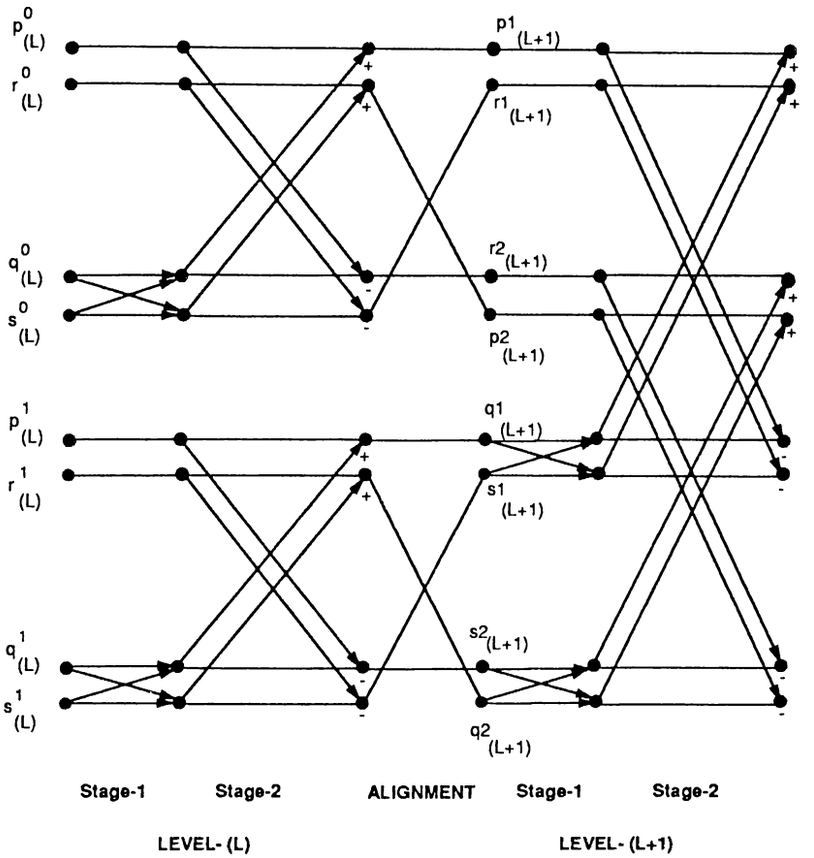

Figure 3.10. Two steps of Restructured Simplified Fast Hartley Transform Butterfly, Type1

and $B2_{(\ell+1)}$ will be allocated to the succesive locations of the X-array if (p, r) points and (q, s) points of both B_ℓ^0 and B_ℓ^1 are initially allocated to the successive locations of the X-array. This structure is valid for both types of butterfly pairs in the proposed scheme, since (p, r) and (q, s) pairs constitute the first two and last two points of B_ℓ^0 and B_ℓ^1 butterflies. That is, if $B^0_{t1(\ell)} = \{p^0_\ell, r^0_\ell, q^0_\ell, s^0_\ell\}$ $= \{i_1, i_1 + 1, i_2, i_2 + 1\}$ and $B^1_{t1(\ell)} = \{p^1_\ell, r^1_\ell, q^1_\ell, s^1_\ell\} = \{j_1, j_1 + 1, j_2, j_2 + 1\}$ then $B1^1_{t1(\ell+1)} = \{p1_{(\ell+1)}, r1_{(\ell+1)}, q1_{(\ell+1)}, s1_{(\ell+1)}\} = \{i_1, i_1 + 1, j_1, j_1 + 1\}$ and $B2^1_{t1(\ell+1)} = \{p2_{(\ell+1)}, r2_{(\ell+1)}, q2_{(\ell+1)}, s2_{(\ell+1)}\} = \{i_3 + 1, i_3, j_3 + 1, j_3\}$. Similarly, if $B^0_{t2(\ell)} = \{i_1, i_1 + 1, i_2, i_2 + 1\}$ and $B^1_{t2(\ell)} = \{j_1, j_1 + 1, j_2, j_2 + 1\}$, then $B1^1_{t2(\ell+1)} = \{p1_{(\ell+1)}, r1_{(\ell+1)}, q1_{(\ell+1)}, s1_{(\ell+1)}\} = \{i_1, i_1+1, j_1, j_1+1\}$ and $B2^1_{t1(\ell+1)}$ $= \{p2_{(\ell+1)}, r2_{(\ell+1)}, q2_{(\ell+1)}, s2_{(\ell+1)}\} = \{i_3, i_3 + 1, j_3, j_3 + 1\}$. This important feature of the proposed re-structuring scheme can be exploited to avoid the fragmentation of the q and s points of *type*-1 FHT butterflies during the FHT

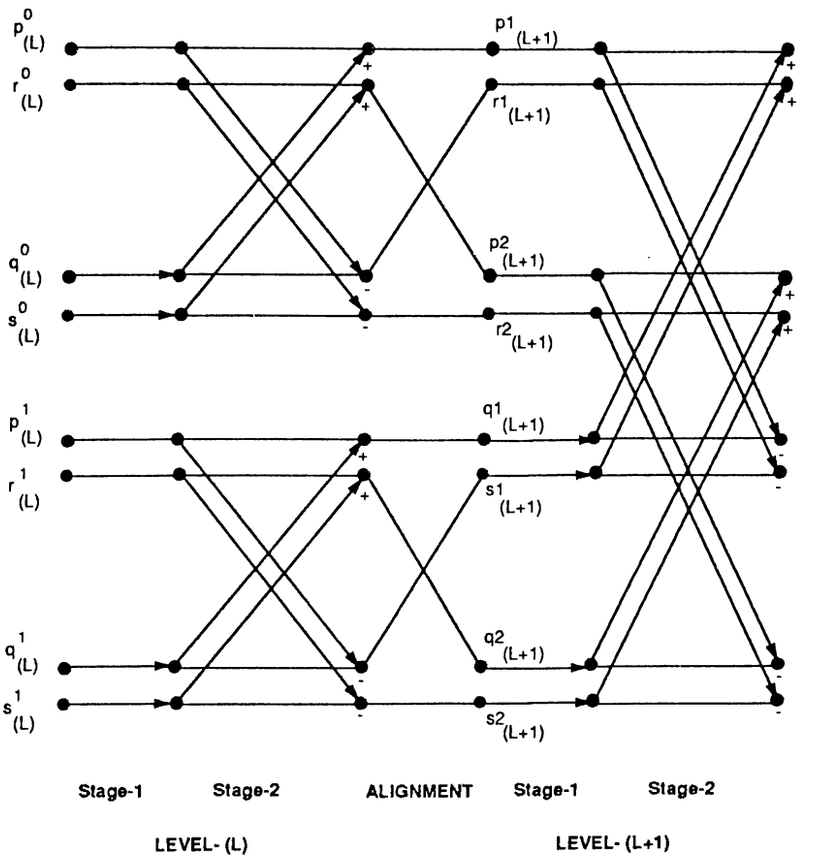

Figure 3.11. Two steps of Restructured Simplified Fast Hartley Transform Butterfly, Type2

implementation.

In the original FHT algorithm, 4-point FHT computations start at level $\ell = 1$ which contains only *type*-2 FHT butterflies. Note that, (p, r) points and (q, s) points of all FHT butterflies at level-1 are already allocated to the successive locations of the X-array. Hence, if the proposed re-structuring is applied starting from level-1, then (p, r) points and (q, s) points of all FHT butterflies at the following levels will be allocated to the successive locations of the X-array. Fig. 3.12 illustrates the computational flow-graph for the re-structured 32-point FHT algorithm. As is seen in Fig. 3.12, the *type*-1 butterfly pair ({X[18], X[19], X[22], X[23]}, {X[26], X[27], X[30], X[31]}) at level-2 constitute the *type*-1 butterfly pair, ({X[18], X[19], X[26], X[27]}, {X[22], X[23],

X[30], X[31]}) at level-3. Similarly, *type*-2 butterfly pair ({X[16], X[17], X[20], X[21]}, {X[24], X[25], X[28], X[29]}) at level-2 constitute the (*type*-2, *type*-1) butterfly pair ({X[16], X[17], X[24], X[25]}, {X[24], X[25], X[28], X[29]}) at level-3. As is also seen in Fig. 3.12, the proposed restructuring does not disturb the block structure of the original FHT algorithm. Furthermore, the proposed restructuring brings regularity and symmetry to the in-block allocation structure of the FHT butterflies. The following paragraph explains the regular allocation structure of $2^{\ell-1} = 2^{\ell+1}/4$ FHT butterflies in each block at level-ℓ ($\ell = 1, 2, \ldots, n-1$).

In each block, $2^{\ell-1}$ consecutive FHT point pairs in the first and second halves constitute the (p, r) and (q, s) pairs, respectively of the FHT butterflies involved in that block. Consecutive FHT point pairs in each half are ordered regularly such that k^{th} ($0 \leq k \leq 2^{\ell-1}$) pairs in the first and second halves constitute the (p, r) and (q, s) pairs of the same FHT butterfly, respectively. The first pairs ($k = 0$) in each half constitute the only *type*-2 FHT butterfly involved in that block. The following $\ell - 1$ consecutive pairs ($k = 1, 2, \ldots, 2^{\ell-1} - 1$) in each half constitute ($2^{\ell-1} - 1$) *type*-1 FHT butterflies involved in that group. However, the last ($2^{\ell-2} - 1$) consecutive pairs ($k = 2^{\ell-2} + 1, \ldots, 2^{\ell-2} - 1$) in each half hold the (p, r) and (q, s) pairs in the reverse order (i.e., as $\{r, p\}$ and $\{s, q\}$). These reversed pairs belong to the second *type*-1 butterflies $B2_{t1(\ell)} = \{p2_\ell, r2_\ell, q2_\ell, s2_\ell\} = \{i_3 + 1, i_3, j_3 + 1, j_3\}$ generated from *type*-1 $(B^0_{t1(\ell-1)}, B^1_{t1(\ell-1)})$ butterfly pairs in the previous level $(\ell - 1)$.

For example, in a 32-point restructured FHT algorithm (See Fig. 3.12) 4-tuplets {0, 1, 8, 9}, {2, 3, 10, 11}, {4, 5, 12, 13}, {7, 6, 15, 14} constitute the $2^{3-1} = 4$ FHT butterflies involved in block $G^0_{(3)} = \{0\text{-}15\}$ at level-3. Note that, first butterfly {0, 1, 8, 9} is the only *type*-2 butterfly involved in $G^0_{(3)}$. Also note that, (p, r) and (q, s) pairs of only the last *type*-1 butterfly {7, 6, 15, 14} = $\{p_3, r_3, q_3, s_3\}$ are hold in reverse order in the X-array since $2^{3-2} - 1 = 1$. As is seen in Fig. 3.12, this *type*-1 butterfly is the second butterfly generated by the *type*-1 butterfly pair ({2, 3, 6, 7}, {10, 11, 14, 15}) in the previous level ($\ell = 2$).

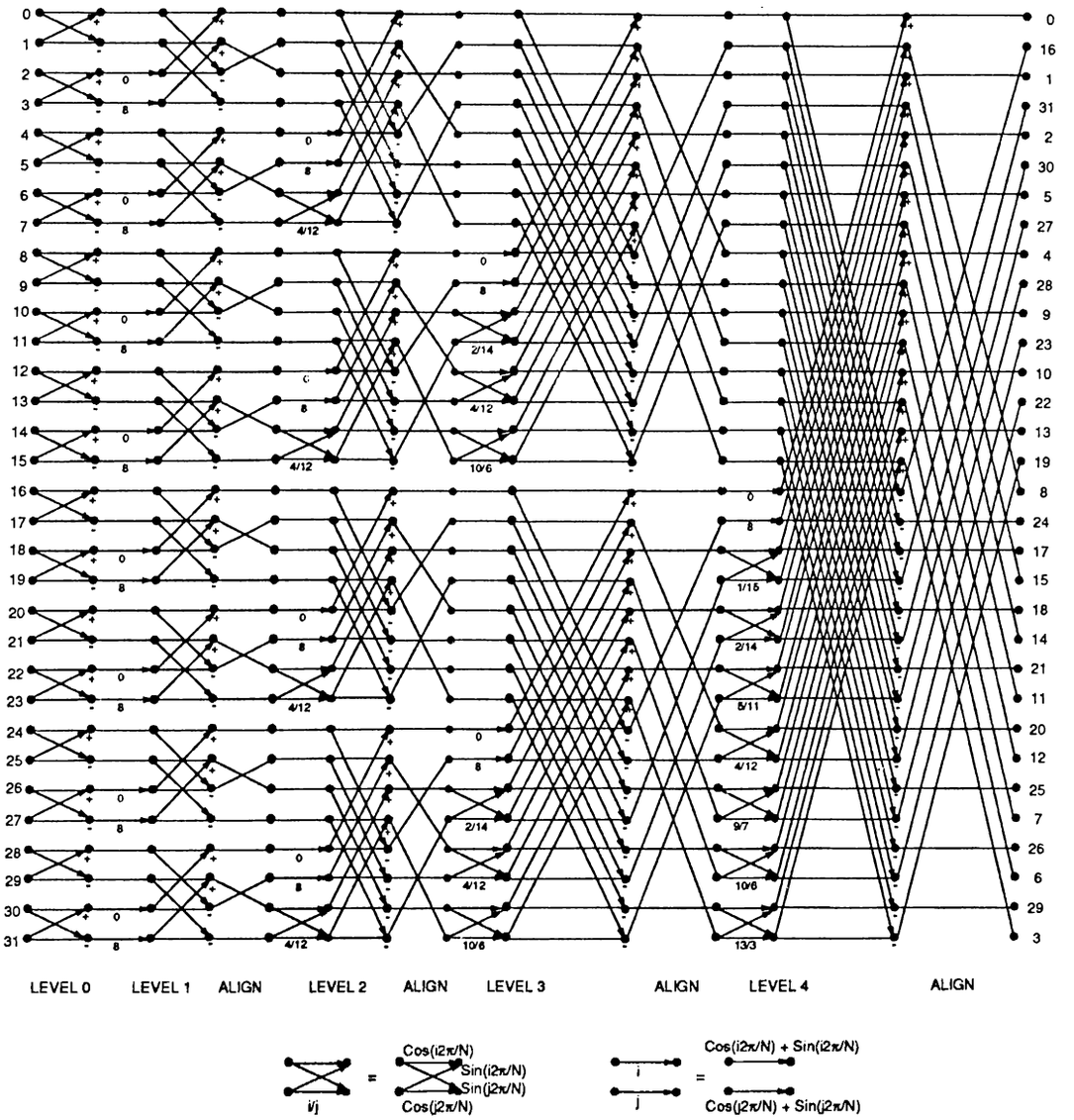

Figure 3.12. Restructured Fast Hartley Transform for N=32 points

Prog. 3.2 illustrates the C-like pseudo-code of the restructured FHT algorithm for the computation of N-point FHT. Note that, Prog. 3.2 has a very similar structure compared to Prog. 3.1 since both programs exploit the same block structure of the FHT computations at each level. However, the assignment statements for p, q, r, s indices are different in Prog. 3.2 due to the restructuring. Furthermore, Eq. 3.9 and Eq. 3.10 are used in Prog. 3.2 instead of Eq. 3.4 and Eq. 3.5, respectively in order to realize the internal allignment operation for each butterfly computation. However, the last $(2^{\ell-2}-1)$ iterations of the inner *for-loop* in the $SEQFHT\ell$ function needs extra attention since last $(2^{\ell-2}-1)$ (p,r) and (q,s) pairs of each level-ℓ block are hold in reverse order in the X-array. A carefull analysis of Eq. 3.4 reveals the symmetry between the computations of p and r points and q and s points of *type*-1 FHT butterflies. That is, correct values for the *type*-1 FHT butterfly will also be computed if we interchange p with r, q with s and i with j in Eq. 3.4. Note that, *qtemp* will hold the correct value of *stemp* and vice versa in this case. This symmetry in *type*-1 FHT butterfly computations is exploited in Prog. 3.2 as follows:

The first two lines in the inner *for-loop* computes the p, r, q and s indices of *type*-1 butterflies involved in a particular block assuming a proper ordering of the FHT points in (p,r) and (q,s) pairs. Hence, during the first $2^{\ell-2}$ iterations, p, r, q and s variables refer to the correct FHT points (i.e., p_ℓ, r_ℓ, q_ℓ, and s_ℓ respectively) in the X-array. However, during the last $2^{\ell-2}-1$ iterations, p, r, q and s indices refer to r_ℓ, p_ℓ, s_ℓ, and q_ℓ points, respectively, in the X-array. Thus, this scheme implicitly achieves the interchange of p with r and q with s in Eq. 3.4. The interchange of the Cos/Sin factors (i.e., interchange of i and j in Eq. 3.4) is also achieved implicitly during construction of the Cos/Sin factor index tables prior to the execution of the program. Hence, correct values for $s_{(\ell+1)}, p_{(\ell+1)}, r_{(\ell+1)}$, and $q_{(\ell+1)}$ are computed effectively in the last four assignment statements of the inner *for-loop*, respectively. However, these values are stored into $X(s), X(q), X(r)$ and $X(p)$ due to the interchange between p and r, and q and s index values. Hence, $p_{(\ell+1)}$ and $s_{(\ell+1)}$ values are effectively swapped (instead of $r_{(\ell+1)}$ and $q_{(\ell+1)}$) during the alignment operation. Recall

```
/* Input in bit-reversed order in X[0 ... N-1]   */
/* Output in scrambled order in X[0 ... N-1]     */
n := log2 N
for (i :=0; i < N − 1; i := i + 2) do
    temp := X(i+1)
    X(i + 1) := X(i) − temp
    X(i) := X(i) + temp
endfor
for (ℓ :=1; ℓ < n; ℓ + +) do
    Call SEQFHTℓ(X,CaSfac,N,ℓ)
endfor

SEQFHTℓ(X,CaSfac,N,ℓ)
    for (i := 0; i < N/2^(ℓ+1); i++) do
        p := i × 2^(ℓ+1);  q := p + 2^ℓ;  r := p + 1;  s := q + 1;
        qtemp := X(q)
        stemp := X(s)
        X(q) := X(r) + stemp
        X(s) := X(r) − stemp
        X(r) := X(p) − qtemp
        X(p) := X(p) + qtemp
        for (j := 2; j < 2^ℓ; j:=j+2) do
            p := i × 2^(ℓ+1) + j;   q := p + 2^ℓ;
            s := q + 1;   r := p + 1;
            qtemp := Cfac1 × X(q) + Sfac1 × X(s)
            stemp := Cfac2 × X(s) + Sfac2 × X(q)
            X(q) := X(p) − qtemp
            X(s) := X(r) + stemp
            X(p) := X(p) + qtemp
            X(r) := X(r) − stemp
        endfor
    endfor
SEQFHTℓ
```

Program 3.2 : Restructured Sequential N-pt FHT Algorithm

that, $r_{(\ell+1)}$ and $q_{(\ell+1)}$ values are swapped during the alignment operations involved in the first $2^{\ell-2}$ iterations of the *for-loop*.

In fact, computation of the last $2^{\ell-2}-1$ *type*-1 butterflies in consecutive blocks corresponds to the combination of the following kind of *type*-1 butterfly pairs in the restructured FHT algorithm; $(B^0_{t1(\ell)}, B^1_{t1(\ell)})$, where $B^0_{t1(\ell)} = \{p^0_\ell, r^0_\ell, q^0_\ell, s^0_\ell\} = \{i_1+1, i_1, i_2+1, i_2\}$ and $B^1_{t1(\ell)} = \{p^1_\ell, r^1_\ell, q^1_\ell, s^1_\ell\} = \{j_1+1, j_1, j_2+1, j_2\}$. These butterfly pairs will have the following allocation structure $B^0_{t1(\ell)} = \{p^0_\ell, r^0_\ell, q^0_\ell, s^0_\ell\} = \{i_2+1, i_1, i_1+1, i_2\}$ and $B^1_{t1(\ell)} = \{p^1_\ell, r^1_\ell, q^1_\ell, s^1_\ell\} = \{j_2+1, j_1, j_1+1, j_2\}$, in the X-array, after the completion of the alignment operation realized in Prog. 3.2. Recall that, $(B^0_{t1(\ell)}, B^1_{t1(\ell)})$ pair always constitute the $(B1_{t1(\ell+1)}, B2_{t1(\ell+1)})$ pair at the next level such that $B1_{t1(\ell+1)} = \{p^0_\ell, s^0_\ell, p^1_\ell, s^1_\ell\}$ and $B2_{t1(\ell+1)} = \{r^0_\ell, q^0_\ell, r^1_\ell, q^1_\ell\}$. Hence, the structure of the $(B1_{t1(\ell+1)}, B2_{t1(\ell+1)})$ pairs will have the following structures $B1_{t1(\ell+1)} = \{p1_{(\ell+1)}, r1_{(\ell+1)}, q1_{(\ell+1)}, s1_{(\ell+1)}\} = \{i_2+1, i_2, j_2+1, j_2\}$ and $B2_{t1(\ell+1)} = \{p2_{(\ell+1)}, r2_{(\ell+1)}, q2_{(\ell+1)}, s2_{(\ell+1)}\} = \{i_1, i_1+1, j_1, j_1+1\}$ in the X-array. For example, the *type*-1 ({7, 6, 15, 14}, {23, 22, 31, 30}) butterfly pair at level-3 constitute the *type*-1 ({15, 14, 31, 30}, {6, 7, 22, 23}) butterfly pair at the next level. It should be noted here that, the computational flow graph given in Fig. 3.12 is constructed according to alignment structure realized in Prog. 3.2. It is clear that, $(B2_{t1(\ell+1)}, B1_{t1(\ell+1)})$ pairs generated during the last $2^{\ell-2}-1$ iterations will have the same spatial structure compared to the $(B1_{t1(\ell+1)}, B2_{t1(\ell+1)})$ pairs generated during the first $2^{\ell-2}$ iterations of the inner *for-loop*. Hence, Prog. 3.2 maintains the regular and symmetrical features of the reconstructed FHT algorithm without disturbing the simplicity and the regularity of programming.

Note, that in Fig. 3.12, output is not in *normal* order, but in slightly scrambled manner. As is mentioned in Chapter 2, the output order is not very much important in applications where an inverse transformation is also needed at the end. Due to the nature of these algorithms, one can always find an inverse algorithm with input in scrambled order and ouput in $bit-reversed$ order.

The straightforward static mapping scheme can also be applied to the restructured FHT algorithm. Fig. 3.13 shows the static mapping of the restructured FHT algorithm on a 2-dimensional hypercube. As is seen, at each level, (during the last d-levels), 2 concurrent exchange operations are necessary. The first exchange is required after the completion of stage-2 computations, in order to gather (p, r) and (q, s) pairs. The second exchange is necessary, in order to align (p, r, q, s) pairs as described earlier. However, in this case, the alignment operations can not be included into the FHT butterfly computations, and should be handled separately. The volume of exchange operations are M and $M/2$ for each communication respectively. As is seen, although load balance is again lost during the last d-levels, both the volume and number of communications are decreased by a considerable amount. Furthermore, these $2d$ concurrent exchange operations are all between nearest neighbors.

In the following paragraphs, it will be shown that, dynamic mapping scheme that was given for perfect-load balance FFT algorithm, can also be applied to the restructured FHT algorithm. This dynamic mapping scheme, when applied to the restructured FHT algorithm, achieves both perfect-load balance, d-concurrent exchange communications (nearest neighbor) with a volume of $N/2P$=$M/2$ FHT points.

As indicated earlier, straightforward mapping scheme Fig. 3.13 already maintains perfect load balance during the first $(n-d)$ levels. Hence, this mapping is maintained during the first $(n-d)$ levels of the perfect-load balance FHT algorithm. As is seen in Fig. 3.13, during the last d-levels, one half of the processors hold only updated values for the (p, r)-points and the other half hold only the updated values for the (q, s)-points of *type*-1 FHT butterflies. This *static* mapping scheme is altered, similar to FFT, at the very beginning of each level of the last d levels. At the very beginning of each level, each processor holding only updated values for the (q, s)-points ($N/2P$ FFT points) exchange one half of its (q, s)-points with the $N/2P$ (p, r)-points of its neighbor processor which holds all the (p, r)-points of its butterfly pairs of that stage of

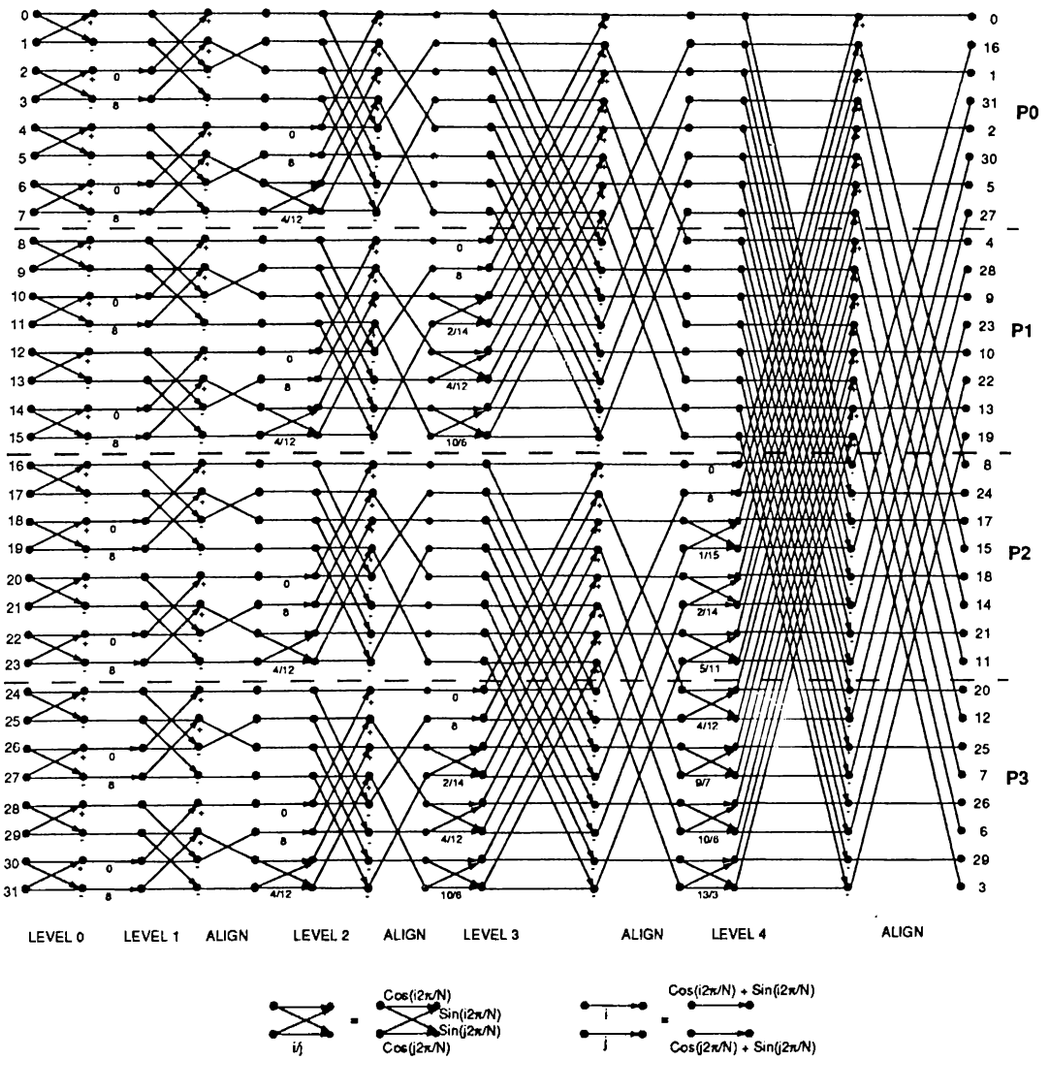

Figure 3.13. Static Mapping of Restructured Fast Hartley Transform for N=32 points on a 2-dimensional hypercube

the algorithm and vice versa. With the exchange operation, processors effectively exchange the responsibility of the further *FHT* computations associated with those exchanged *FHT* points and hold equal number of p, r, q and s points.

This *dynamic* mapping scheme is which is very similar to Fig. 2.4 is illustrated in Fig. 3.14 for a 32-point *FHT* on a 2-dimensional hypercube. The pseude-code for the node-program of the proposed parallel *FHT* algoithm is given in Prog. 3.3. Again a C-like notation is used in Prog. 3.3 The pseudo-code, given in Prog. 3.3, is for the last d-levels, since the first $(n-d)$ levels are exactly same as in Prog. 3.2. As is seen, in Fig. 3.14 and Prog. 3.3, each processor exchanges either the first half or the second half of its local X-array in place by simply checking the k^{th} bit value of its processor index, where k denotes the channel over which the exchange operation is to be performed on that level. Due to the *dynamic* mapping scheme, each processor performs *simplified FHT* butterfly computations on local (p,r) and (q,s) pairs seperated by $2^{m-1} = N/2P$ after the exchange operations at each level of the last d-levels. Although (p,r) and (q,s) pairs are partioned equally among the processors, the type of *FHT* butterflies that they form, are not. In other words, each processor, during the last d-levels, contains $M/4$ *FHT* butterfly pairs. The first butterfly pair in each processor is a *type*-2 *FHT* butterfly if the k^{th} bit of that prorocessor is equal to "0". Otherwise, it is a *type*-1 *FHT* butterfly as well as the remaining $M/4 - 1$ *FHT* butterflies. So, during the last d-levels, one half of the processors which have a zero in their k^{th} bit of their processor number, have only 1 *type*-2 and $M/4 - 1$ *type*-1 *FHT* butterflies, while the other half have $M/4$ *type*-1 *FHT* butterflies. Hence, this difference in the computations is solved simply by the second *if*-statement before the inner *for-loop*.

Note that, since the alignment operations are confined within *FHT* butterfly pairs, no extra communication is necessary for the swap operations. Dynamic mapping scheme proposed for the *FFT* algorithm also achieves perfect load balance for the restructured *FHT* algorithm. Furthermore, the number and volume of concurrent exchange communications are reduced to d and $N/2P$, respectively. Also, nearest neighbor communications are also achieved

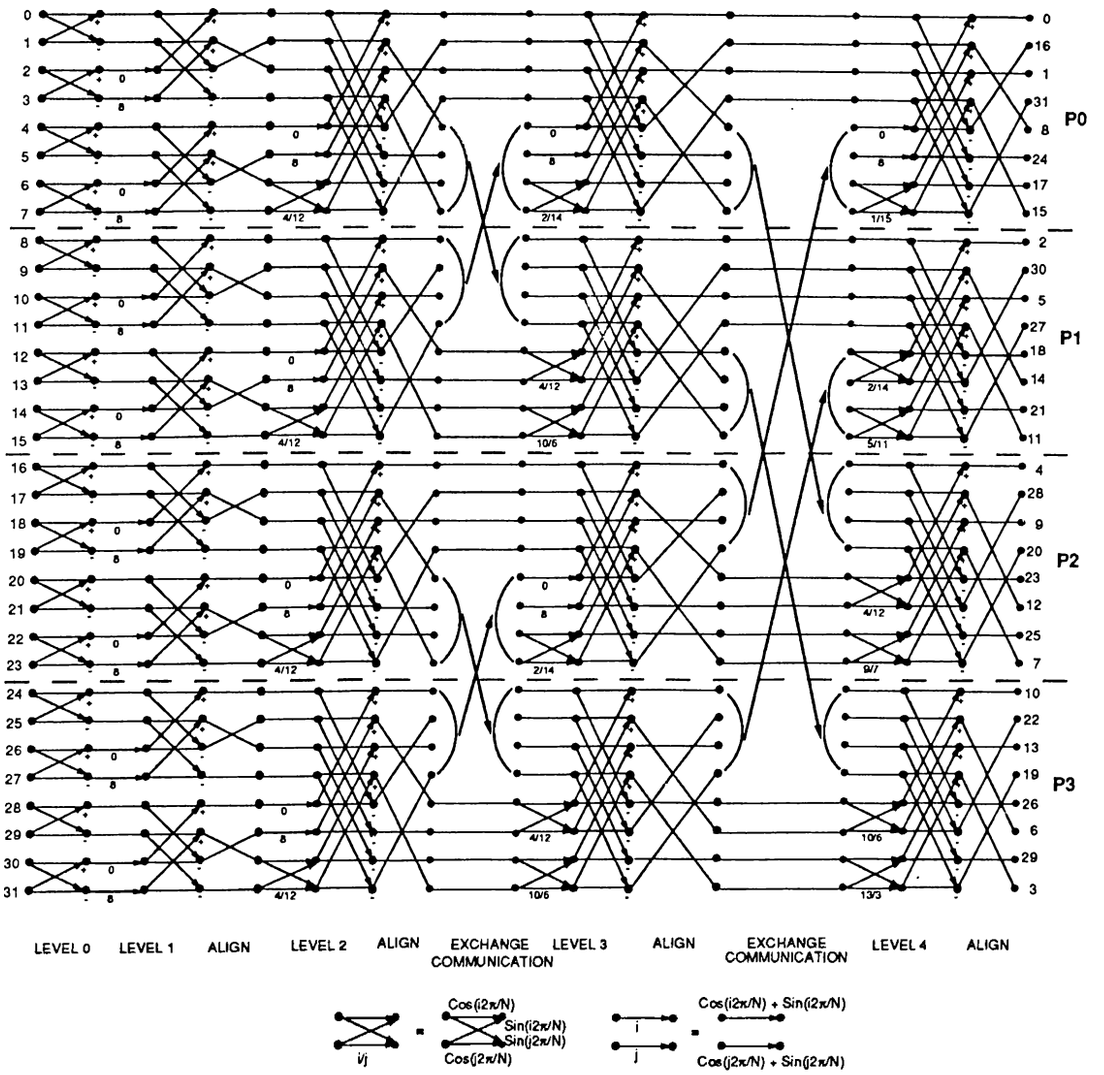

Figure 3.14. Dynamic Mapping of Fast Hartley Transform for N=32 points on a 4-node hypercube

```
/* Computations over the next d bits; d-concurrent exchange phase */
for (ℓ := n−d; ℓ < n; ℓ++) do
    k := ℓ−(n−d);
    if ((k^th bit of mynode) = 0) then do
        csend from (X(i): i=M/2, M/2 − 1, ..., M − 1) to dnode
        crecv into (X(i): i=M/2, M/2 − 1, ..., M − 1) from dnode
    else
        csend from (X(i): i=0, 1, ..., M/2 − 1) to dnode
        crecv into (X(i): i=0, 1, ..., M/2 − 1) from dnode
    endif
    if ((k^th bit of mynode) = 0) then do
        p := 0;  q := M/2;  r := p + 1;  s := q + 1;
        qtemp := X(q);  stemp := X(s)
        X(q) := X(r) + stemp
        X(s) := X(r) − stemp
        X(r) := X(p) − qtemp
        X(p) := X(p) + qtemp
    else
        p := 0;  q := M/2;  r := p + 1;  s := q + 1;
        qtemp := Cfac1 × X(q) + Sfac1 × X(s)
        stemp := Cfac2 × X(s) + Sfac2 × X(q)
        X(q) := X(p) − qtemp
        X(s) := X(r) + stemp
        X(p) := X(p) + qtemp
        X(r) := X(r) − stemp
    endif
    for (i := 1; i < M/2; i++) do
        p := 2;  q := p + 1;  r := p + M/2;  s := r + 1;
        qtemp := Cfac1 × X(q) + Sfac1 × X(s)
        stemp := Cfac2 × X(s) + Sfac2 × X(q)
        X(q) := X(p) − qtemp
        X(s) := X(r) + stemp
        X(p) := X(p) + qtemp
        X(r) := X(r) − stemp
    endfor
endfor
```

Program 3.3 : Perfect Load Balance FHT Algorithm on a hypercube with P processors (last d-levels)

with no fragmentary message passing. Hence the parallel complexity of the proposed scheme is,

$$T_{P3} = [\frac{10N}{4P}(\log_2 \frac{N}{P} - 1) - \frac{N}{P} + 4]\, t_{calc} + [\frac{10N}{4P} \log_2 P]\, t_{calc} + [t_{su} + \alpha \frac{N}{2P} t_{tr}] \log_2 P \quad (3.11)$$

where t_{su} is the message startup overhead and t_{tr} is the time taken for the transmission of a real floating-point word (4-bytes). The definition of α was given in Chapter 2. The first term is for the complexity of the first $(n - d)$ steps, while the second term is for the last d concurrent exchange phase and the last term is for the communication overhead.

As is seen in Fig. 3.14, the order of output is scrambled in blocks of $(N/2P)$ with respect to Fig. 3.13, similar to FFT. The sum of indices of the output is N for consecutive two pairs, except for the first pair which is $N/2$, similar to the scrambled output order of the restructured FHT algorithm Prog. 3.2

3.4 Experimental Results

All the programs written, have been coded in C language and run on a 32-node iPSC/2 hypercube multicomputer for various $N = 2^n$ data sizes, $128 \leq N \leq 64K$. Table 3.1 displays the sequential timing results of FFT and the original FHT, Prog. 2.1 and Prog. 3.1. A carefull analysis on the timing results show that, FHT algorithm takes nearly one half of the time that a similar FFT algorithm takes, which is as expected. The restructured FHT algorithm is also implemented. The performance of the original and the restructured FHT algorithms are observed to be nearly the same.

Parallel FHT algorithm with static mapping is not implemented for reasons of loosing load balance, high number and volume of data exchanges as well as multi-hop communications. On the other hand, the proposed parallel FHT algorithm with dynamic mapping (Prog. 3.3), is implemented. High efficiency

N	**Prog.2.1**	**Prog.3.1**
64	11.1	4.6
128	25.5	11.0
256	58.0	25.8
512	129.8	59.4
1K	287.7	134.3
2K	631.5	300.0
4K	1375.6	662.1
8K	2978.0	1450.0
16K	6480.2	3156.6
32K	13964.2	6933.0
64K	29791.7	14972.0

Table 3.1. Sequential timing results of FFT and FHT, Prog.(2.1) and Prog.(3.1) (msec).

and speedup values are obtained for Prog. 3.3.

Fig. 3.15 displays the speed up curve for Prog. 3.3. As one can notice, in Fig. 3.15, nearly linear speed-up is achieved for large N. Fig. 3.16 displays the efficiency curve for Prog. 3.3. As is seen in Fig. 3.16, efficiency remains over % 85 when $N/P \geq 512$ FHT points are mapped to an individual processor of the hypercube. The reason why high efficiencies like % 90 when $N/P \geq 512$ are not obtained is due to the small granularity of FHT algorithm compared to that of FFT.

3.5 Conclusion

The Fast Hartley Transform which is a promising alternative to the Fast Fourier Transform is parallelized for hypercube connected multicomputers. The proposed restructured sequential FHT algorithm brings regularity and symmetry to the computation of FHT. The given restructured algorithm, when parallelized as in the case of FFT, achieves both *perfect-load* balance and nearest neighbor communication, requires only d-exchange communications, has a volume of communication $N/2P$ and eliminates fragmentary message passing.

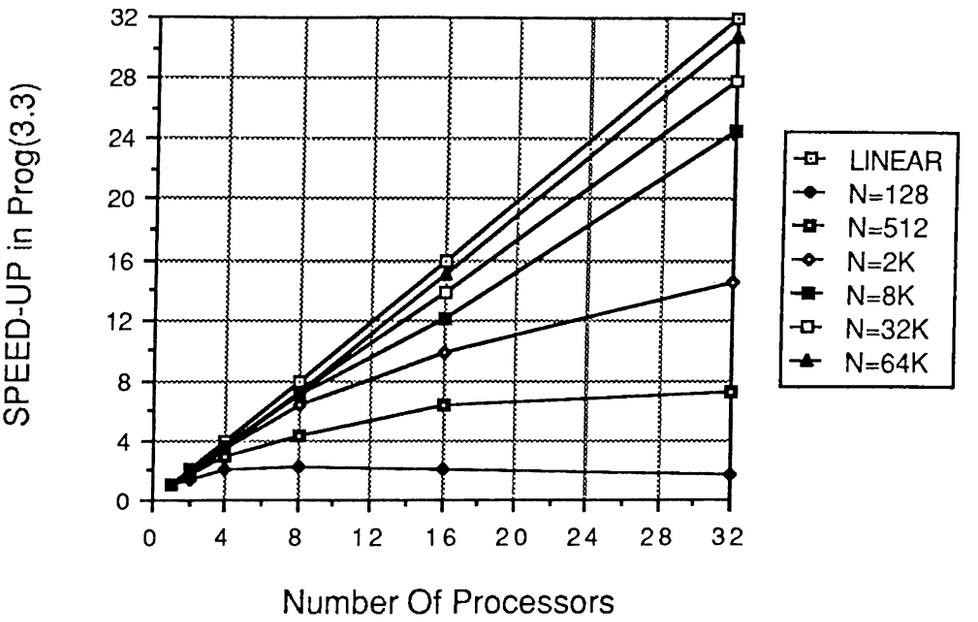

Figure 3.15. Speedup curve for Prog.(3.3).

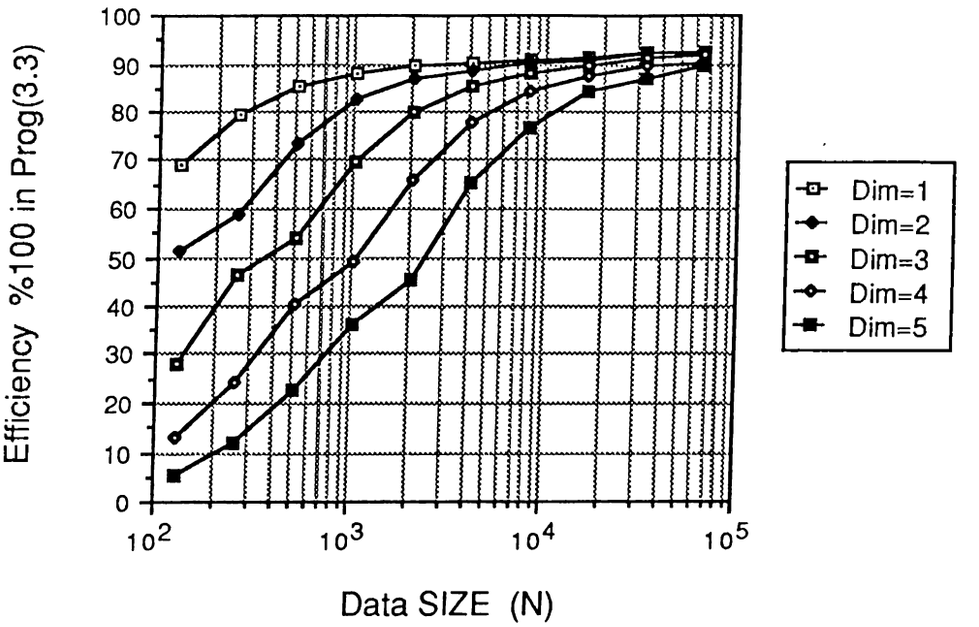

Figure 3.16. Efficiency curve for Prog.(3.3).

The FHT algorithm requires only one half the memory space that of a similar FFT algorithms and computes in one half the time of a similar FFT algorithm. The algorithm also uses *in-place* computations and table lookup scheme for the calculation of trigonometric values. The proposed parallel FHT algorithm is implemented on Intel's iPSC/2 hypercube multicomputer with 32 processors. Again high efficiency values are obtained for small size problems even if the granularity of FHT algorithm was less than that of FFT algorithm. As in the case of FFT, similar algorithms that overlap communication with computations are also possible which may increase the performance of the proposed algorithms.

4. The Fast Cosine Transform

4.1 Introduction

Since it's introduction in 1974 [20], *Discrete Cosine Transform* has found wide applications in image and signal processing in general and mostly in data compression, filtering and pattern recognition. When compared to other orthogonal transforms, it is found to compare closely to Karhunen-Loeve transform which is known to be optimum with respect to variance distribution, estimation using the mean-square error criterion and rate distortion function. Although KLT is optimal, there is no general algorithm that enables its fast computation. This is why *Discrete Cosine Transform* is so popular.

The DCT of a data sequence $x(n)$, $(n = 0, 1, \ldots, N-1)$ is defined as,

$$\begin{aligned} X(0) &= \frac{\sqrt{2}}{N} \sum_{n=0}^{N-1} x(n) \\ X(k) &= \frac{2}{N} \sum_{n=0}^{N-1} x(n) Cos(\frac{(2n+1)k\pi}{2N}) \end{aligned} \tag{4.1}$$

where $k = 1, 2, \ldots, N-1$.

As well as FFT and FHT, there exists also a fast transform for DCT [25] referred in this work as *Fast Cosine Transform* or simply FCT. There exists several algorithms for the computation of FCT and these algorithms are generally based on two approaches.

1. Direct factorization of DCT matrix
2. Indirect factorization of DCT matrix

Direct factorization algorithms [21, 22, 23, 24, 25, 26] are always superior to indirect factorization algorithms. On the other hand indirect factorization algorithms which make use of FFT [20, 30] or FHT [27, 31] to compute FCT are simpler to implement because they use such well known transforms. There also exists recursive algorithms [24] which can be counted among the direct factorization methods.

Among these approaches, Lee's algorithm [25] which is a direct factorization algorithm is discussed and Malvar's algorithm [31] which is an indirect factorization approach using FHT is parallelized and implemented.

In Section 4.2.1 Lee's sequential Fast Cosine Transform is discussed and implemented. In Section 4.2.2 Malvar's sequential Fast Cosine Transform algorithm is discussed and implemented. In Section 4.3, Malvar's Fast Cosine Transform is parallelized and implemented on a 32-node iPSC/2 hypercube. In Section 4.4, experimental results and performance of Malvar's parallelized FCT algorithm is given.

4.2 Sequential FCT Algorithms

Lee's FCT algorithm [25], which is known to be the best sequential algorithm using the direct factorization methods [16], will be discussed in the first subsection. On the other hand, Malvar's FCT algorithm [31] which is an indirect factorization method using FHT, but not as efficient as Lee's algorithm, is discussed in the second subsection.

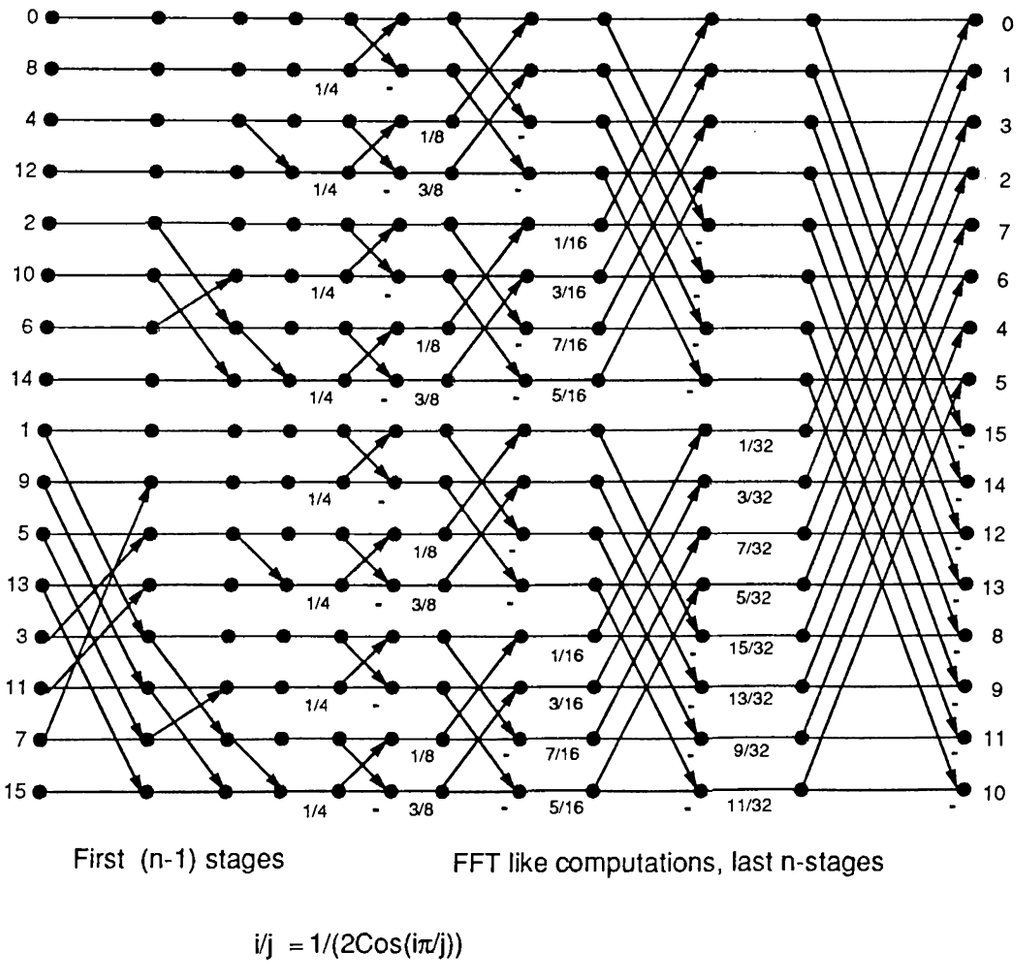

Figure 4.1. Computational Flow Graph of 16-point FCT (Lee's method).

4.2.1 Lee's Sequential FCT Algorithm

Figure 4.1 illustrates the flow-graph of a 16-point FCT using Lee's approach is given. As is seen, the computations can be easily divided into two phases of length $n-1$ and n steps respectively, where $n = \log_2 N$ and N=input length. The first phase, contains only addition operations, while the second phase resembles an FFT like structure where the W factors are replaced with Cos factors. As is seen in Figure 4.1, the input is in *bit-reversed* order similar to FFT, and the output is in a scrambled manner as discussed below.

```
/* Input in bit-reversed order in X[0 ... N-1]   */
/* Output in scrambled order in X[0 ... N-1]    */
n := log2 N
for (k :=0; k < n − 1; k++) do
    Call ADDk(X,N,k)
endfor
Call SEQFFTk(X,Cfac,N)


ADDk(X,N,k)
    for (i := 0; i < N; i++) do
        Call ALIGNk(X,Y,2^(n−k))
        for (j := 0; j < 2^(n−k−2); j++) do
            p := 2^(n−k−2) × 2 + i × 2^(n−k) + j
            q := p + 2^(n−k−2)
            X(q) := X(q) + X(p)
            X(p) := X(p) + Y(j)
        endfor
    endfor
end ADDk
```

Program 4.1 : Sequential N-pt FCT Algorithm (Lee's Method)

The order of the output sequence $x(k)$ is generated by starting with the set (0, 1) and adding the prefix "0" to each element and than obtaining the rest of the elements by complementing the existing ones. This process results in the set (00, 01, 11, 10) and by repeating it, one can obtain (000, 001, 011, 010, 111, 110, 100, 101). Thus, the output sequence (0, 1, 3, 2, 7, 6, 4, 5) is obtained for N=8.

As is seen in Figure 4.1, the computations are of type additions only. In other words, at the k^{th} stage, where $0 \leq k < n-1$, necessary operations are only additions on FCT points whose $(n-k)^{th}$ bits are ones. At the second phase, where $n-1 \leq k < 2n-1$, the computations resemble an FFT like structure except the W_N^r factors which are now replaced with C_j^i factors $(C_j^i = \frac{1}{2Cos(\frac{i\pi}{j})})$.

The algorithm given in Prog. 4.1, shows the two phase operation clearly.

In Prog. 4.1, the first *for-loop*, corresponds to the section of first $n-1$ steps, while the stage, $SEQFFTk$, corresponds to the section of last n steps which resembles FFT like structure very much. The $ADDk$ function is used for the computation of first $n-1$ steps. The $ALIGNk$ function that is used inside the $ADDk$ function reorders some of the data before entering into the computations in each stage. $ALIGNk$ copies elements from $X()$ to $Y()$ in the reverse order with blocks of length $1, 2, \ldots, N/2^{n-2-k}-1$. Given a sequence, (0,1,2,3,4,5,6,7) the output from the $ALIGNk$ function is as follows: (-,7,5,6,1,2,3,4). The q points that are going to be added with p points, have to be reordered inside the $ADDk$ function, since the computations can not be done in-place otherwise. Total number of points that are going to be ordered in each stage, where $0 \leq k \leq n-1$, is equal to $N/4-1$.

For the computation of last n steps, Prog. 2.1, is used. Indeed Prog. 2.1 is now called with a pointer to $Cfac$ instead of $Wfac$ and the input type of Prog. 2.1 is also changed from *complex* to *real*.

The complexity of Prog. 4.1 is as follows,

$$\begin{aligned} T_{P1} &= ([\frac{N}{2}(\log_2 N - 2) + 1] + [\frac{3N}{2}(\log_2 N])\ t_{calc} \\ &= (2N\log_2 N - N + 1)\ t_{calc} \end{aligned} \tag{4.2}$$

where t_{calc} is the time taken for a real-floating point operation, addition or multiplication. The first term comes from the first $n-1$ levels of the algorithm, while the second term comes from the last n levels of the algorithm.

4.2.2 Malvar's Sequential FCT Algorithm

In this section, an indirect factorization method named after Malvar [31] is discussed. Malvar has showed that there exists a relation between FCT and FHT. Furthermore Malvar stated that, it is possible to compute Fast Cosine Transform of an input sequence through Fast Hartley Transform by using the

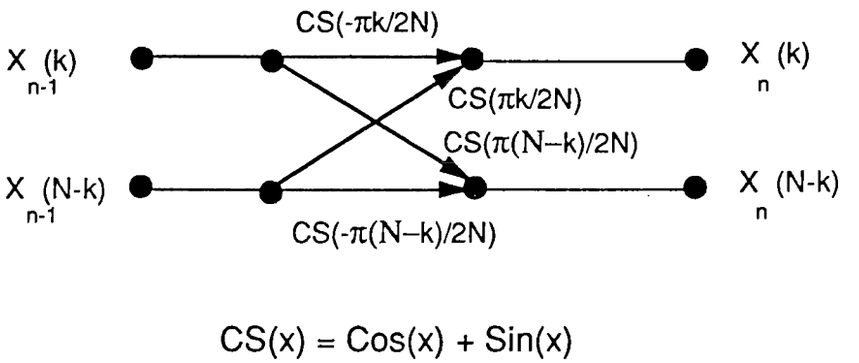

Figure 4.2. Basic FCT Butterfly

following relation,

$$\begin{bmatrix} C(k) \\ C(N-k) \end{bmatrix} = \frac{1}{2} \begin{bmatrix} cas(\frac{-\pi k}{2N}) & cas(\frac{\pi k}{2N}) \\ cas(\frac{\pi(N-k)}{2N}) & cas(\frac{-\pi(N-k)}{2N}) \end{bmatrix} \begin{bmatrix} H(k) \\ H(N-k) \end{bmatrix} \quad (4.3)$$

for $k = 1, 2, \ldots, N/2 - 1$, where $cas(\theta) = cos(\theta) + sin(\theta)$. While, $C(0)$ and $C(N/2)$ are equal to,

$$\begin{bmatrix} C(0) \\ C(\frac{N}{2}) \end{bmatrix} = \begin{bmatrix} H(0) \\ \frac{1}{\sqrt{2}} H(\frac{N}{2}) \end{bmatrix} \quad (4.4)$$

$H(k)$ is the Fast Hartley Transform of input data, and $C(k)$ is the Fast Cosine Transform of input data, computed using the Fast Hartley Transform. As is seen, the relation between FHT and FCT is very simple and straightforward. Necessary operations to compute FCT after the computation of FHT are only 2 multiplication and 1 addition per FCT point.

Figure 4.2 shows the basic FCT butterfly that is described in Eq. 4.3. As is seen, the sum of indices of points that enter into the basic FCT butterfly is always N except for $X(0)$ and $X(N/2)$ which is $N/2$. Necessary operations to compute the basic FCT butterfly are only 4 multiplications and 2 additions.

Since the basic FCT butterflies are applied after the last level of Figure 3.6, (Sequential FHT) algorithm, and the outputs of FHT are also in *normal* order,

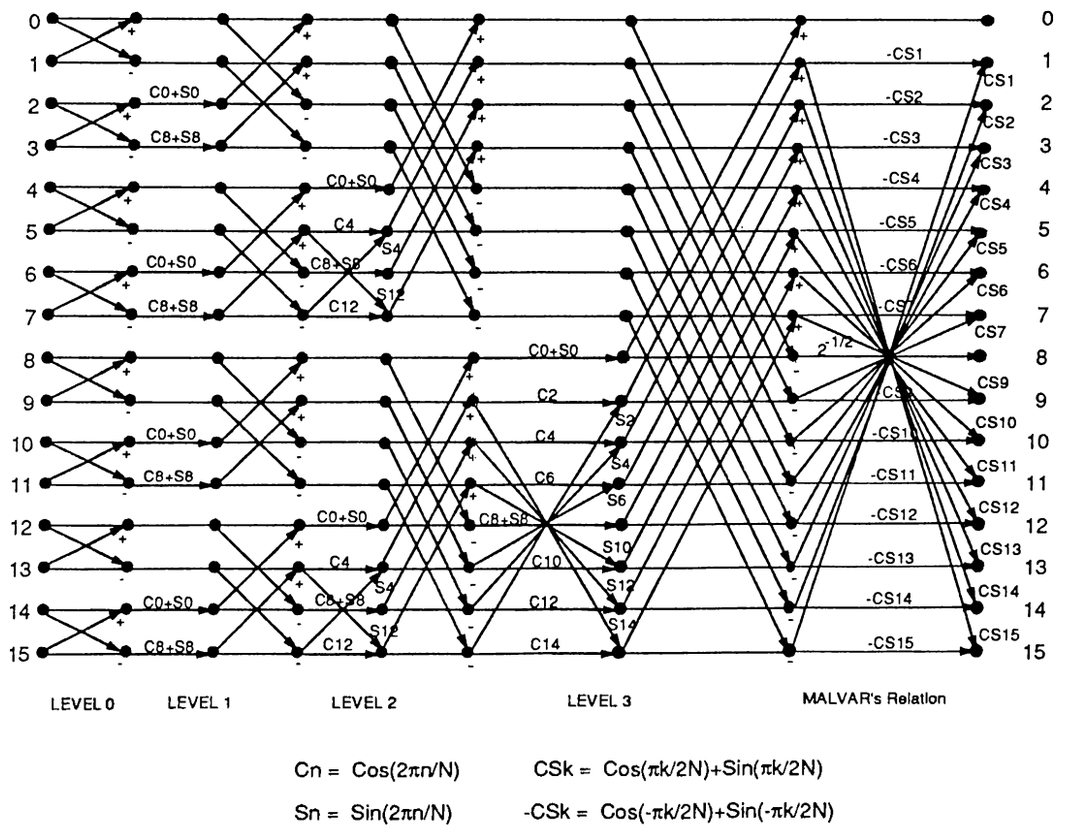

Figure 4.3. Computational Flow Graph of 16-point FCT (Malvar's method).

no further indexing is necessary. Basic FCT butterflies can easily be appended at the end of Figure 3.6.

In Figure 4.3, flow-graph of 16-point FCT computations using the *basic* FCT butterfly is given. The first $n = \log_2 N$ steps are for the computation of FHT using the algorithm presented in Figure 3.6. The last step is the Malvar's equation for FCT 4.3. As is seen, the outputs are in *normal* order, while the inputs are again in *bit-reversed* order. In other words, *basic* FCT butterflies does not change the output order.

```
/* Input in bit-reversed order in X[0 ... N-1]   */
/* Output in Y[0 ... N-1]    */
/* First Compute Fast Hartley Transform by using Prog.3.1     */
Call SEQFHT(X,CaSfac,N)

/* Use Malvar's relation between FHT and FCT */
for  (k := 0; k < N/2; k++)
    Y(k) := X(k) × CaSfac(-k) + X(N-k) × CaSfac(k)
    Y(N-k) := X(k) × CaSfac(N-k) + X(N-k) × CaSfac(k-N)
endfor
```

Program 4.2 : Sequential N-pt FCT Algorithm (Malvar's method).

In Prog. 4.2, pseudo-code for the sequential FCT algorithm using the Malvar's relation is given. As is seen, the first part is a call to the sequential FHT algorithm that was developed in Chapter 3. The last part, *for-loop*, is for the computation of Malvar's equation given in Eq. 4.3. Basic FCT butterflies is applied at the n^{th} level. As is seen, indexing of *basic* FCT butterflies is very straightforward.

The complexity of Malvar's FCT algorithm can be expressed as follows,

$$T_{P2} = T_{FHT} + [3N - 5]t_{calc} \tag{4.5}$$

where T_{FHT} is the time complexity of sequential Fast Hartley Transform given in Chapter 3 and the last term is for the complexity of Malvar's Eq. 4.3.

Replacing T_{FHT}, complexity expression becomes,

$$\begin{aligned} T_{P2} &= ([2.5N\log_2 N - \frac{9}{2}N - 6] + 3N - 5)t_{calc} \\ &= [2.5N\log_2 N - \frac{3}{2}N - 11]t_{calc} \end{aligned} \tag{4.6}$$

4.3 Perfect Load Balance FCT Algorithm

As mentioned in [25], Lee's algorithm achieves to be the best sequential algorithm. However Lee's algorithm involves strong interdependences during the

first $n-1$ levels and can not be efficiently parallelized. Lee's algorithm, when parallelized, requires at least $d-1$ communications with a volume of N/P and d communication with a volume of $N/2P$, making a total of $2d-1$. Furthermore some of the first $d-1$ communications are between non-neighbor nodes; multi-hop messages. Also it is very hard to obtain *perfect load balance* during the first $d-1$ steps. In other words, during the first $d-1$ steps, only processors with $(d-1-k)^{th}$ bits $= 1$, do computations while the others $(d-1-k)^{th}$ bits $= 0$, stay idle. In the second phase, since the computations resemble an FFT like structure, a dynamic mapping that was given in Chapter 2, can also be applied. But since, perfect load balance and nearest neighbor communications can not be achieved and too many communications are required, it is clear that it will not achieve high efficiencies when parallelized. The described method for the parallelization of Lee's algorithm achieves a time complexity of,

$$\begin{aligned} T_{Parallel-Lee} &= (\frac{N}{P}\log_2\frac{P}{2})\ t_{calc} + \frac{N}{2P}(\log_2\frac{N}{P} - 2)\ t_{calc} + \\ &\quad [t_{su} + t_{hop} + \alpha\frac{N}{P}t_{tr}]\ \log_2\frac{P}{2} + (\frac{3N}{2P}\log_2\frac{N}{P})\ t_{calc} + \\ &\quad \log_2 P\ (t_{su} + \alpha\frac{N}{2P}t_{tr}) + (\frac{3N}{2P}\log_2 P)\ t_{calc} \end{aligned} \tag{4.7}$$

where α is the parameter defined in Section 2.3.1, t_{su} is the message startup overhead and t_{tr} is the time taken for the transmission of a real floating-point word (4 bytes) between two neighbor processors, t_{hop} is the message routing time needed for the communications between non-neighbor processors as is defined in [32]. The first three terms are for the first $n-1$ steps of the algorithm while the last three terms are for the last n-steps of the algorithm.

As discussed earlier, Malvar's FCT algorithm uses FHT during the computations. So a faster FHT computation also means a faster FCT algorithm. The restructured FHT algorithm and its dynamic mapping scheme proposed in Chapter 3 can be exploited for an efficient parallelization of the Malvar's FCT algorithm. In the rest of this section, Malvar's FCT algorithm will be investigated for parallelization and it will be shown that, Malvar's FCT algorithm can outperform Lee's algorithm in the parallel case.

In Figure 4.4, dynamic mapping of a 32-point FCT on a 4-node hypercube .

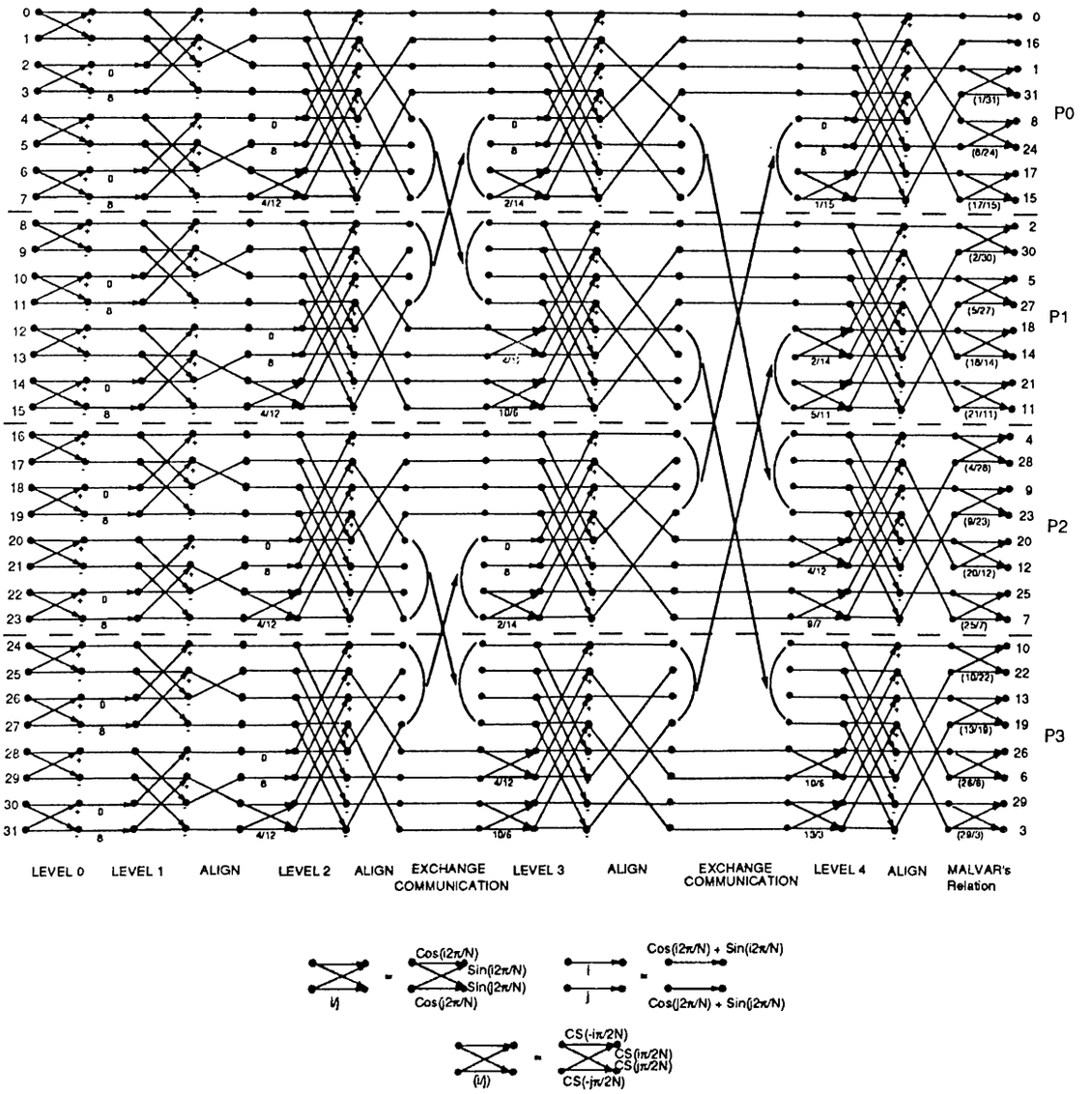

Figure 4.4. Dynamic mapping of 32-point FCT on a 4-processor hypercube (Malvar's method).

is given. As is seen in Figure 4.4, the first $n = \log_2 N$ steps is actually the parallel FHT algorithm given in Prog. 3.2. The last step is for the computation of FCT from FHT using the Malvar's relation given in Eq. 4.3. It can be easily seen that, each FHT pair that is going to be used in Malvar's relation, $X(k)$ and $X(N-k)$, is located to the the same processor. Furthermore, these points are all in consecutive placesn the X-array. In other words, Malvar's relation neither requires extra communications nor indexing overhead.

In Prog. 4.3, pseudo-code for the parallel FCT algorithm using the Malvar's relation and parallel FHT algorithm proposed in Section 3.3.1 is given. As is seen in Prog. 4.3, a simple call operation (PARFHT) is done at the very beginning, to the parallel FHT algorithm (Prog. 3.3) that was developed in Chapter 3. After the computation of FHT by the dynamic mapping scheme that was described before, Malvar's relation is applied by a simple *for-loop*. The last *for-loop*, as mentioned before requires no-communication and also achieves *perfect-load* balance. So the overall algorithm also achieves *perfect-load balance*, nearest neighbor communications and require only a total of d concurrent exchange communications steps. The volume of each exchange communication operation is only $N/2P$ FCT points.

Time complexity of the parallel FCT algorithm given in Prog. 4.3 is therefore,

$$T_{P3} = T_{PARFHT} + \frac{3N}{P} t_{calc} \tag{4.8}$$

where T_{FHT} is the time complexity of parallel Fast Hartley Transform algorithm given in Chapter 3 and when put in the above relation the time complexity relation becomes as follows,

$$\begin{aligned} T_{P3} &= 2.5[\frac{N}{P}\log_2 N]t_{calc} - [\frac{9}{2}\frac{N}{P} + 6]t_{calc} + \\ &\quad [t_{su} + \alpha\frac{N}{2P}t_{tr}]\log_2 P + \frac{3N}{P}t_{calc} \\ &= [2.5\log_2 N - 1.5P - 6]t_{calc} + [t_{su} + \alpha\frac{N}{2P}t_{tr}]\log_2 P \end{aligned} \tag{4.9}$$

As is seen from Eq. 4.9 and Eq. 4.7, Malvar's FCT algorithm can easily outperform Lee's algorithm when parallelized.

```
/* Input in bit-reversed order in X[0 ... N-1]  */
/* Output in Y[0 ... N-1]    */
/* First Compute Fast Hartley Transform by using Prog.3.2    */
Call PARFHT(X,CaSfac,N,P)

/* Use Malvar's relation between FHT and FCT */
M := N/P;
for  (i := 0; i < M/2; i:=i+2)
    Y(i) := X(i) × CaSfac(-k) + X(i+1) × CaSfac(k)
    Y(i+1) := X(i) × CaSfac(N-k) + X(i+1) × CaSfac(k-N)
    Y(i+M/2) := X(i+M/2) × CaSfac(k) + X(i+M/2+1) × CaSfac(-k)
    Y (i+1+M/2) := X(i+M/2) × CaSfac(k-N) + X(i+M/2+1) × CaSfac(N-k)
endfor
```

Program 4.3 : Parallel N-pt FCT Algorithm on a hypercube with P processors (Malvar's method).

4.4 Experimental Results

All programs presented in this chapter have been coded in C language and run on a 32-node iPSC/2 hypercube multicomputer for various $N = 2^n$ data sizes, $128 \leq N \leq 64K$. Speedup and efficiency curves are obtained with respect to Lee's sequential FCT algorithm.

A comparison of Lee's and Malvar's sequential algorithms, Prog. 4.1 and Prog. 4.2, is given in Table 4.1. As is seen in Table 4.1, Prog. 4.1 is slightly faster than Prog. 4.2. Although Lee's, FCT algorithm is better in the sequential case, it can not achieve *perfect-load* balance during the first $(n-1)$ stages, furthermore there exists non-nearest neighbor communications with a volume of N/P during this first phase. On the other hand, Malvar's relation about FCT is very suitable to apply on the algorithm that was proposed in Section 3.3.1.

Fig. 4.5 shows the speedup curve for Prog. 4.3 for various data sizes, $256 \leq$

N	*Prog.* 4.1	*Prog.* 4.2
128	12.1	12.5
256	28.0	28.2
512	62.3	63.1
1K	137.5	139.8
2K	302.7	306.9
4K	660.9	668.7
8K	1435.1	1447.6
16K	3100.2	3124.5
32K	6740.3	6785.6
64K	14440.0	14542.1

Table 4.1. Timing results (msec) for sequential FCT algorithms, Prog. 4.1 and Prog. 4.2.

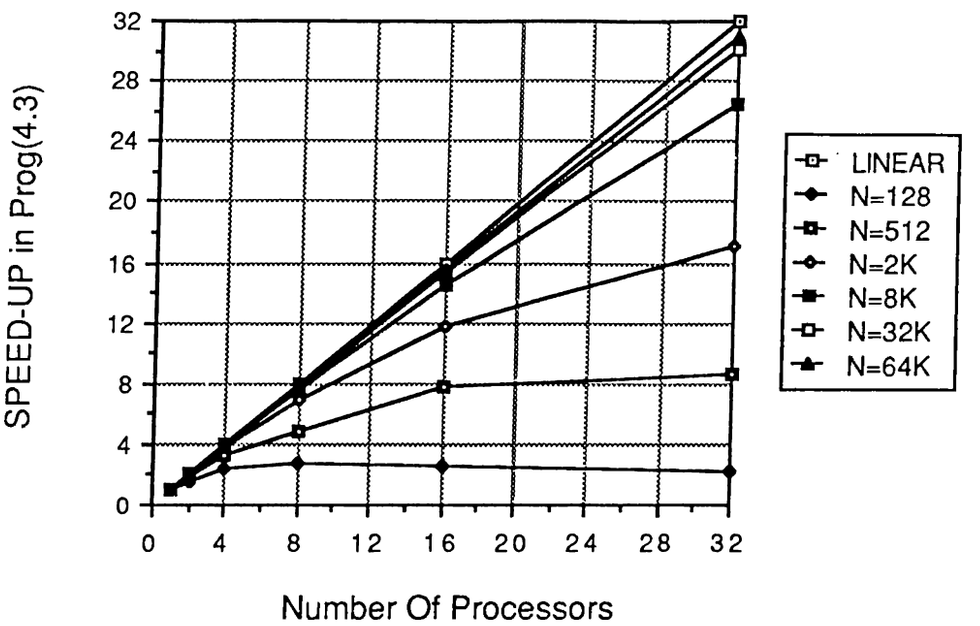

Figure 4.5. Speedup curve for Prog. 4.3.

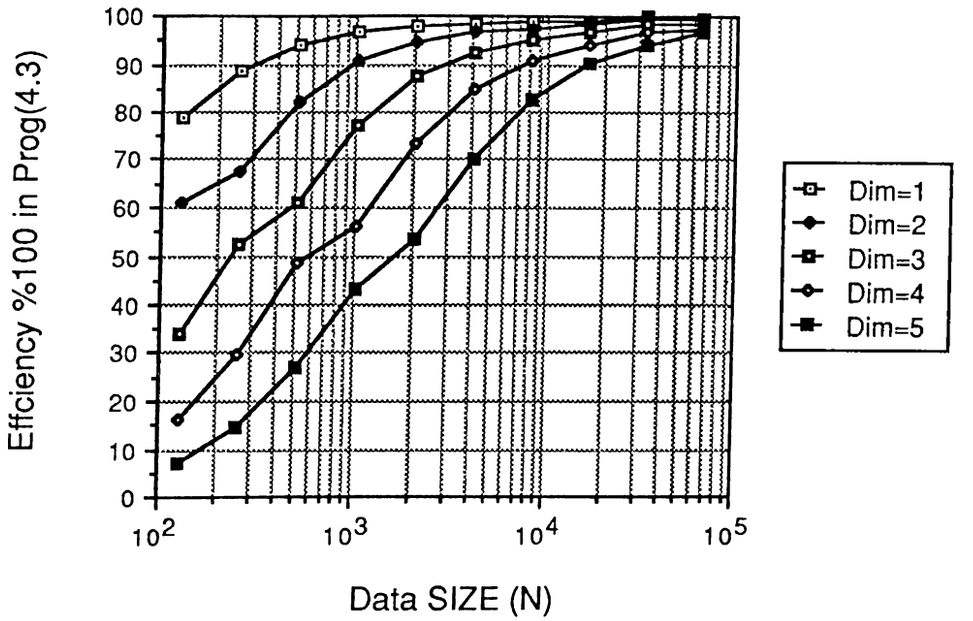

Figure 4.6. Efficiency curve for Prog. 4.3.

$N \leq 64K$. As is seen in Fig. 4.5, almost linear speedup is obtained even for large cube dimensions. Figure 4.6 shows the efficiency curve for Prog. 4.3 for different cube dimensions and various data sizes. As is seen, over 90 % effciencies are obtained when $N/P \geq 512$ FCT points are mapped to individual iPSC/2 nodes. Furthermore parallel FCT algorithm (Prog. 4.3) implemented in this work shows a performance which is close to that of FFT and FHT programs presented in Chapter 2 and Chapter 3 respectively. Although Lee's FCT algorithm is not parallelized for the reasons that are explained before, it is clear enough that Lee's algorithm will not perform better than the given parallel FCT algorithm.

4.5 Conclusion

In this chapter, two different FCT algorithms are discussed for parallelization The performance of best sequential FCT algorithm, Lee's algorithm, is shown to be slightly faster compared to Malvar's algorithm. Lee's algorithm is shown to be not suitable for parallelization on a medium-to-coarse grain multicomputer. On the other hand, Malvar's algorithm, which computes FCT through

FHT, is efficiently parallelized by exploiting the efficient parallel FHT algorithm proposed in Chapter 3. Proposed parallel FCT algorithm achieves *perfect-load* balance as well as nearest neighbor communications with a number of d concurrent communications with a volume of $N/2P$ FCT points. Proposed parallel FCT algorithm is implemented on Intel's iPSC/2 hypercube multicomputer with 32 processors. High efficiency values are obtained for the proposed parallel FCT algorithm. Furthermore, overlapping of communications with computations is also possible as in the case of FFT and FHT.

5. Conclusion

In this thesis, parallelization of several algorithms in the field of Digital Signal Processing (DSP) are investigated. The investigated DSP algorithms are Fast Fouirer Transform (FFT), Fast Hartley Transform (FHT) and Fast Cosine Transform (FCT). These algorithms are parallelized for distributed-memory, message-passing multiprocessors implementing the hypercube interconnection topology.

All parallel algorithms achieve the following for an N-point problem on a d dimensional hypercube with $P = 2^d$ processors.

- maintain perfect load-balance
- minimize communication overhead
  - eliminate fragmentary message passing
  - allow nearest neighbor communication
  - require only d concurrent exchange communication steps
  - necessitate only $N/2P$ data point exchange in each communication
- can overlap communications with computations
- achieve regular computational patters so that they can also be implemented on SIMD type hypercube multicomputers

The proposed parallel algorithms are implemented on Intel's iPSC/2 hypercube multicomputer with 32 processors. High efficiency and almost linear

speed-up values are obtained for even small size problems. Experimental results show that hypercube topology is very suitable for the parallel computation of FFT, FHT and FCT. These problems can be computed in real-time on large size hypercube multicomputers.

Bibliography

[1] J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Math. Comput.*, Vol. 19, pp. 297-301, April 1965.

[2] I.J. Good "The relationship between two fast fourier transform", *IEEE Transactions on Computers*, C-20, pp.310-317, 1971.

[3] S. Winograd "On Computing the Discrete Fourier Transform", *Math. Comp.*, 32, pp.175-199, 1978.

[4] G. Goertzel "An Algorithm for the evaluation of Finite Trigonometric Series", *Amer. Math. Monthly*, 65, pp.34-35, 1968.

[5] A.V. Oppenheim "Digital Signal Processing," *Prentice Hall International.*

[6] L. Bomans, and D. Roose, "Benchmarking the iPSC/2 hypercube multiprocessor," in *Concurrency : Practice and Experience*, Vol. 1(1), pp. 3-18, September 1989.

[7] S. R. Walton, "Performance of the One-Dimensional Fast Fourier Transform on the Hypercube," in *Second Conference on Hypercube Multiprocessors*, Knoxville, pp. 530-535, September 1986.

[8] J.P. Zhu, "An Efficient FFT algorithm on Multiprocessors with Distributed Memory," in *The Fifth Distributed Memory Computing Conference*, Vol. 1(2), pp 358-363, January 1990.

[9] D. W. Walker, "Portable Programming within a Message-Passing Model: the FFT as an Example," in *Third Conference on Hypercube Concurrent Computers and Applications*, Pasadena, CA, pp. 1438-1450, January 1988.

[10] C. Aykanat, F. Ozguner, F. Ercal, and P. Sadayappan, "Iterative Algorithms for Solution of Large Sparse Systems of Linear Equations on Hypercubes," in *IEEE Transactions on Computers*, vol 37, no. 12, pp. 1554-1568, December 1988.

[11] R.V.L Hartley, "A more symmetrical Fourier analysis applied to transmission problems", *Proceedings of IRE*, vol. 30, pp. 144-150, March 1942.

[12] O. Buneman, "Conversion of FFT's to Fast Hartley Transforms", in *SIAM J. Sci. Stat. Comput.*, vol. 7, No. 2, April 1986.

[13] R.N. Bracewell, "The Fast Hartley Transform", in *Proceedings of the IEEE*, vol. 72, No. 8, August 1984.

[14] Henrik V.Sorensen, Douglas L.Jones, C.Sidney Burrus and Micheal T.Heideman, "On Computing the Discrete Hartley Transform", in *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. ASSP-33, No.4, October 1985.

[15] T. Le-Ngoc and M. Tue Vo, "Implementation and Performance of The Fast Hartley Transform", in *IEEE Micro*, pp. 20-27, October 1989.

[16] Hsieh S.Hou, "The Fast Hartley Transform Algorithm", *IEEE Transactions on Computers*, vol. C-36, No.2, February 1987.

[17] J.D. Villasenor and R.N. Bracewell, "Vector Hartley Transform", in *Electronics Letters*, 17 August 1989, vol.25, No.17, pp.1110-1111.

[18] X. Lin, T.F. Chan and W.J. Karplus, "The Fast Hartley Transform on the hypercube multiprocessors", in *Third Conference on Hypercube Concurrent Computers and Applications*, Pasadena, CA, pp. 1451-1454, January 1988.

[19] H. Hao, "On Fast Hartley Transform Algorithms", in *Proceedings of the IEEE*, vol. 75, No. 7, pp. 951-952, July 1987.

[20] N. Ahmed, T. Natarajan and K.R. Rao, "Discrete Cosine Transform", *IEEE Transactions on Computers*, pp.90-93, January 1974.

[21] Z.Wang, "On Computing the Discrete Fourier and Cosine Transform", *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. ASSP.33 No.4, pp.1341-1344, October 1985.

[22] Z.Wang, "Fast Algorithms for the Discrete W Transform and for the Discrete Fourier Transform", *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. ASSP.32 No.4, pp.803-816, August 1984.

[23] N.Suehiro and M.Hatori, "Fast Algorithms for the DFT and other Sinusoidal Transforms", *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. ASSP.34 No.3, pp.642-644, June 1986.

[24] H.S Hou, "A Fast Recursive Algorithm for Computing the Discrete Cosine Transform", *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. ASSP.35 No.10, pp.1455-1461, October 1987.

[25] B.G. Lee, "A New Algorithm to Compute the Discrete Cosine Transform", *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. ASSP.32 No.6, pp.1243-1245, December 1984.

[26] W.H. Chen, C.H. Smith and S.C. Fralick, "A Fast Computational Algorithm for the Discrete Cosine Transform", *IEEE Transactions on Communications*, Vol. COM.25 No.9, pp.1004-1009, September 1977.

[27] H.S. Malvar, "Fast Computation of the Discrete Cosine Transform and the Discrete Hartley Transform", *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. ASSP.35 No.10, pp.1484-1485, October 1987.

[28] M.A. Haque, "A Two-Dimensional Fast Cosine Transform", *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. ASSP.33 No.6, pp.1532-1539, December 1985.

[29] M. Vetterli and H.J. Nussbaumer, "Simple FFT and DCT algorithms with reduced number of operations", *Signal Processing*, Vol.6, pp.267-278, 1984.

[30] M.J. Narasimha and A.M. Peterson, "On the computation of the Discrete Cosine Transform", *IEEE Transactions on Communications*, Vol. COM.26 No.6, pp.934-936, June 1978.

[31] H.Malvar, "Fast Compuatation of Discrete Cosine Transform Through Fast Hartley Transform", *Electronic Letters*, Vol.22 No.7, pp.352-353, 27 March 1986.

[32] S.R.Seidel and T.E. Schmiermund, "Refining the Communication Model for the Intel iPSC/2",

[33] A.Dervis, K.Oflazer and F.Ercal, "Experiments with Parallel Sorting on iPSC/2 Multicomputer", *International Symposium on Computer and Information Sciences-V*, pp.319-325, vol(1), Nevsehir, TURKEY, 1990.

[34] F.Ercal, A.Dervis and C.Aykanat, "Experimenting with FFT on an iPSC/2 Hypercube", *Communication, Control and Signal Processing*, pp.1691-1696, vol(2),Ankara,TURKEY,1990.

[35] C.Aykanat and A.Dervis, "An Overlapped FFT Algorithm for Hypercube Multicomputers", *International Conference in Parallel Processing*, ICPP-91, U.S.A, 1991.

[36] A.Dervis and C.Aykanat, "Overlapped FFT algorithms on iPSC/2", *International Symposium on Computer and Information Sciences-VI*, ISCIS-6, Side, TURKEY, 1991.